DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF 93940

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PRODUCTION RULE SYSTEMS AS AN APPROACH
TO INTERPRETATION OF
GROUND SENSOR INFORMATION

by

Dennis Michael Jackson

June 1980

Thesis Advisor: D.H. Smith

Approved for public release; distribution unlimited

T195842

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)
DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF 93940

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Production Rule Systems as an Approach To Interpretation of Ground Sensor Information	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; June 1980
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Dennis Michael Jackson	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: June 1980
	13. NUMBER OF PAGES: 269
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Naval Postgraduate School, Monterey, California 93940	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence
Production Rule Systems
Ground Sensor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Knowledge-based production systems, which are based on Artificial Intelligence concepts, offer distinct advantages in the representation, analysis, and correlation of information in certain problem areas. This thesis presents a detailed design and implementation of a prototype knowledge-based production system which interprets information from unattended ground sensors. Background, rationale, system objectives and system structure are discussed. Representation of knowledge and the program control structure are emphasized. Finally, the details of implementation and a discussion of the results obtained are presented.

Approved for public release; distribution unlimited.

Production Rule Systems as an Approach To Interpretation of Ground Sensor Information

by

Dennis M. Jackson
Captain, United States Marine Corps
B.S., United States Military Academy, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1980

ABSTRACT

Knowledge-based production systems, which are based on Artificial Intelligence concepts, offer distinct advantages in the representation, analysis, and correlation of information in certain problem areas. This thesis presents a detailed design and implementation of a prototype knowledge-based production system which interprets information from unattended ground sensors. Background, rationale, system objectives and system structure are discussed. Representation of knowledge and the program control structure are emphasized. Finally, the details of implementation and a discussion of the results obtained are presented.

TABLE OF CONTENTS

I. INTRODUCTION.....8

II. BACKGROUND.....13

A. PREFACE.....13

B. THE MARINE CORPS GROUND SENSOR PROGRAM.....13

1. Purpose.....14

2. Organization and Training.....14

3. Employment Concepts of Ground Sensors.....17

4. Advantages of Remote Sensor Employment....20

a. Timeliness.....20

b. All Weather.....20

c. Full Time.....21

d. Expendability.....21

e. Survivability.....21

f. Covertness.....21

g. Flexibility.....21

5. Disadvantages of Remote Sensor Employment.....21

a. High Cost.....22

b. Limited Discrimination.....22

c. Limited Life.....22

d. Jamming.....22

e. Range.....22

f. Location.....22

g. Enemy Reaction.....22
6. Factors for Consideration in the Employment of Ground Sensor Devices.....23
a. Mission of the Unit Being Supported...23
b. Type of Coverage Desired.....23
c. Terrain Aspects.....23
d. Weather Considerations.....24
e. Enemy Considerations.....24
f. Support Requirements.....24
7. Methods of Employment.....24
8. Remote Ground Sensor Equipment.....25
C. AVAILABLE METHODS OF PROCESSING GROUND SENSOR INFORMATION.....28
III. DESIGN.....31
A. PREFACE.....31
B. GENERAL.....31
1. Structure and Characteristics of Knowledge-based Problem Solving Production Systems.....32
2. Appropiate and Inappropiate Domains for Knowledge-based Production Systems....37
3. Advantages of Knowledge-based Production Systems.....39
4. Disadvantages of Knowledge-based Production Systems.....40
5. Hierarchical Knowledge-based Production

Systems.....41
C. A PROTOTYPE PRODUCTION SYSTEM TO PROCESS GROUND SENSOR INFORMATION.....44
1. Overview.....44
2. Design of the Prototype Knowledge-based Production System.....46
IV. IMPLEMENTATION.....53
A. GENERAL.....53
B. PROGRAM STRUCTURE.....54
C. PROGRAM CAPABILITIES.....61
D. SAMPLE RULE AND COMPUTATION.....62
V. DISCUSSION OF RESULTS.....65
A. OVERVIEW.....65
B. TESTING OF THE PROTOTYPE PRODUCTION SYSTEM.....65
1. Validation Testing.....65
2. Performance Testing.....66
a. Scenario Number One.....67
b. Scenario Number Two.....68
C. PERFORMANCE TEST RESULTS.....69
D. CONCURRENT TESTING.....70
VI. CONCLUSIONS AND RECOMMENDATIONS.....73
A. CONCLUSIONS.....73
B. RECOMMENDATIONS.....75
APPENDIX A SYSTEM PRODUCTION RULES.....80
PROTOTYPE PRODUCTION SYSTEM PROGRAM.....86

DRAWINGS.....261

A. FIGURES.....261

1. Example Sensor Readout.....261

2. Processing Methods.....262

3. Overall System Design.....263

4. Production System Levels.....264

5. System Configuration Before Activation.....265

6. System Configuration After Activation.....266

BIBLIOGRAPHY.....267

INITIAL DISTRIBUTION LIST.....269

I. INTRODUCTION

The ability to collect and transmit information has multiplied several-fold in recent years, in large part due to the advent of the microprocessor and advanced communications technology. This leap in capability has made combat intelligence and combat information more accessable to every level of command. The abundance of information available makes it technically possible for any unit commander to acquire information concerning the status of friendly as well as hostile forces within his area of influence. To exploit this new capability, however, it becomes necessary to develop the means to correlate this information in a meaningful way. By meaningful, it is meant that the essential elements must be culled from the larger body of information, that they be extracted in a timely manner so that the information produced is relevant, and that the information be presented to the user in a manner that he can understand and act upon. As more and more information becomes available at every level, this capability for effective correlation becomes increasingly important.

This thesis represents a limited attempt to investigate and act upon this problem of correlation of information. The nature of the problem itself seems to be pertinent to the

entire military hierarchy, varying only in the scope, context, and the degree of correlation required to produce useful information. It was desirable to limit the investigation to an area where any results produced would be meaningful, and where the information needed to conduct the effort would be readily available. After some investigation, the target of the research and immplementation effort was chosen to be a subset of the Marine Corps ground sensor program.

The United States Marine Corps currently possesses the capability to acquire information through the deployment and use of networks of ground sensors. These sensor devices, of which there are several types, were originally developed for use in Southeast Asia. They are emplaced in a variety of ways, and provide information about their immediate environment once activated. This information is presented to the collector in its raw form. In the case of seismic sensors, for example, which detect vibrations in the earth caused by the passage of an object within the detection radius of the sensor, the information is presented as a series of binary (on or off) activations. The quantity and quality of the information is variable over time; the information may, for example, be corrupted to a varying extent by the environment in which the sensor is placed, by the operating characteristics of the sensor itself, or by jamming. Correlation, which takes place at the collection

site, is presently manual in nature, and is performed by trained operators who compare the output of adjacent sensors, sensor arrays (a collection of sensors logically grouped together to to provide information on a specified area) and sensor fields (a collection of arrays logically grouped to provide information over a larger area) over time. This correlation effort serves as an input to the Marine Corps intelligence system and can be used to provide combat information to the applicable commanders [1].

This thesis focuses on one method of correlating information that seems to be applicable to a wide variety of problems - production systems. A branch of Artificial Intelligence, performance oriented knowledge-based expert production systems provide a means for encoding the expert knowledge of a restricted domain [2]. A performance-oriented production system attempts to display competent behavior in that restricted domain by the use of selected applications of production rules, each of which represent some small piece of knowledge that is applicable to the domain. While perhaps computationally equivalent to an algorithmic approach to the same problem, the production system differs significantly in the manner in which knowledge is represented and applied. Production rules offer a representation of knowledge that is easily accessed and modified, making them quite useful for systems designed for an incremental approach to competence [3]. The knowledge of

the system is extended simply by adding rules that apply to the domain, and the informal knowledge of the expert may be represented and applied in a convenient and useful manner. Representation of knowledge and its application in this manner offers inherent flexibility, and produces behavior similiar to that which a human can produce and understand. In this manner, a production system may offer a feasible approach to a class of problems that would otherwise be very difficult to structure algorithmically, or given a solution, be difficult to explain in a comprehensive manner to the average user.

The thesis is organized into five major sections, excluding the introduction and attachments. The Background section provides information concerning the organization of the Marine Corps ground sensor program, the technology used in the present generation of ground sensors, and processing methods that have been developed in the past or are currently being utilized in this area. This information will hopefully give the reader a feel for the nature of the problem and put the thesis effort in the proper context. The Design section discusses hierarchical performance-oriented production systems in general, their salient features, their structure, and their characteristics. Production systems already in existence that influenced the approach to the system designed in this thesis effort are mentioned. This section includes a discussion of the structure of the

prototype production system developed as a result of the thesis to process and correlate ground sensor information. The objective of the design effort, the concept, the structure, and the design approach are included. The Implementation section describes just that - the implementation of the prototype system in its present form. An example rule and a computation that results from the operation of this rule are presented. The final two sections discuss the results obtained by the limited testing conducted, and the conclusions drawn from the results of the testing. Recommendations are presented for further development, and a discussion of the applicability of further study is included.

II. BACKGROUND

A. PREFACE

This chapter is divided into two major areas. The first area contains information about the Marine Corps ground sensor program. Only that information is presented which allows the reader to place the thesis effort in proper context, give the reader an idea of the scope and complexity of the problem area, or was pertinent to the thesis effort itself. More detailed information on the Marine Corps ground sensor program may be obtained by referring to ref. 1, 4, and 5. The second area contains information about previous development efforts in this area, and current methods of processing and correlating ground sensor information. It is hoped that this information will give the reader an understanding of the "state of the art" of processing information in this area, and the background that he may better understand why a production system was the method chosen for investigation.

B. THE MARINE CORPS GROUND SENSOR PROGRAM

This section is divided into eight subsections, each covering a facet of the Marine Corps ground sensor program. The purpose, organization and training, employment concept,

advantages and disadvantages, factors for consideration in employment, methods of emplacement, and the equipment considered in this thesis are detailed. As noted before, information that is more detailed than that presented may be found in the above cited references.

1. Purpose

The Marine Corps sensor surveillance capability was designed and is organized to meet Marine Corps requirements in support of amphibious operations. Data acquired from the use of ground sensors (a detailed description of such a sensor is given in section B.8 of this chapter) can result in the identification, monitoring, and/or the targeting of hostile elements that might otherwise go undetected. Collation of sensor surveillance acquired data with that acquired by other means is intended to result in a significantly improved intelligence capability [1]. As such, the Marine Corps ground sensor program is an integral part of the total intelligence collection capability of the Marine Corps.

2. Organization and Training

Sensor assets within the Marine Corps are controlled, maintained, and operated by the sensor control and management platoon (SCAMP). One SCAMP is allocated per division; each SCAMP functions under the cognizance of the

division intelligence officer. The mission of a SCAMP is to establish the capability for remote sensor surveillance employment in amphibious operations, to support contingency operations, and to conduct sensor surveillance training and testing as required [1]. The SCAMP platoon is also tasked to support the Tactical Surveillance Center (TSC), which is a self-contained, mobile, air transportable operations center designed to provide a capability for monitoring and evaluating sensor derived data. The SCAMP is organized with the capability of providing sensor employment squads and teams to Regiments and Battalions, respectively, subordinate to the Division. The SCAMP is also capable of supporting combat support units; i.e. the Division Reconnaissance battalion, and the Artillery regiment. Each member of a sensor employment squad or team is trained to implant the various classes of sensor devices found in the Marine Corps inventory, monitor these devices, read out transmitted data from the sensors, and analyze the result. Thus configured, the SCAMP represents centralized control of all sensor assets within the Division, but stands ready to support subordinate units and fulfill their particular sensor requirements as the situation dictates [5].

In its present configuration, the SCAMP is capable of simultaneously monitoring up to seven hundred twenty different sensors of the various types that are contained in the Marine Corps inventory. This is accomplished through the

operation of the TSC, mentioned previously. The TSC is actually a series of four separate shelters, each capable of stand-alone operation and each able to monitor two hundred forty different sensors simultaneously. Each shelter is designed to be operated by a sensor employment squad. Each shelter contains four recording devices (to be described later), each of which is capable of monitoring sixty sensors simultaneously and is operated by one Marine. As a modular structure, the structures of the TSC may be tasked out to subordinate units as required, or may be colocated to function as a unit [4].

The training of SCAMP personnel is accomplished by various means, most of which (technical training in electronics, for example) comes from the basic military occupational speciality that the Marine originally came from, and will not be further discussed. Sensor-specific training, however, is of interest; for the Marine in the SCAMP platoon, it comes primarily from two sources. Initial training in the sensor field is usually received courtesy of the Army through its Unattended Ground Sensors course, given at the U. S. Army Intelligence Center at Fort Huachuca, Arizona. This school is designed as an entry level course (i.e. is very basic in nature), but it is the only formal training a Marine sensor operator is likely to get. After graduation from the Army school, the Marine is assigned to one of the three SCAMP's that are currently active. This

represents his entry into the second phase of the training, which is likely to last for the duration of his career in the Marine Corps - on-the-job and unit level training. Each SCAMP has its own training program, which is tailored to the situation that each particular SCAMP finds itself in, plus the desires of the various levels of commanders that are able to influence the SCAMP platoon commander. The tailored training programs and a high turnover rate, especially among key personnel such as platoon commanders (who directly influence the scope, style, and content of training programs) serve to add elements of inconsistency to the various training programs.

3. Employment Concepts of Ground Sensors

The current concept of employment of ground sensor assets calls for remote sensors to be emplaced before, during, and after assault operations where enemy activity is known or expected. Employment is generally classified by the organizational level at which the sensors and their resulting information are utilized, and is broken down into three different types of employment classes: types I, II, and III methods of employment [1]. In the case of every type of employment, locations for employment and types of sensors to be employed are mission dependent and are preselected so as to optimize the receipt of information, given the level

of resources available (i.e. the number of sensors, their types, and their monitoring capability).

Sensors employed in a type I mode are locally employed by subordinate elements of the amphibious task force, down to and including the battalion level. Data from the sensors is immediately evaluated and acted upon locally. Control is decentralized; sensors in this mode of operation are normally used for local perimeter defense, listening posts, local security patrols, ambush operations, and local targeting for organic (in-house) supporting arms (mortars and artillery) [1].

Type II employment of sensors, on the other hand, centralizes control of sensor assets to directly support the amphibious force/landing force to provide timely information about enemy forces and movements within the amphibious objective area. Type II employment is characterized by utilization of sensor assets at a higher tactical level. Representative uses for sensors in this mode of employment include preassault area surveillance to assist in the selection of landing zones, surveillance along large scale defense perimeters, and surveillance and targeting of enemy approach and withdrawl routes to and from the beachhead [1].

Type III employment is the highest level of employment, and is essentially strategic in nature. Control of sensor assets is passed to higher levels in the chain of command. In this mode, sensors are normally employed in the

area of interest, where immediate combat action is not contemplated, but where continuous surveillance is considered necessary. Examples of sensor employment in this mode include employment along border areas through which enemy units may pass, enemy airfields, harbors, and base areas, or nominally neutral zones [1].

On the most basic level, sensors are normally employed in groups (hereafter referred to as arrays). The sensors within an array represent a logical grouping, designed to provide as much information as possible about a given area. The sensors provide multiple sources of information so that classification and tracking may be initiated on suspected targets. The configuration and composition of a particular array is dependent upon the situation in which an array is emplaced, but as a general rule, arrays may be broken down into two distinct classifications by method of employment: the trail array, and the area array. The trail array is a grouping of sensors that is essentially linear in nature; the array is usually employed along a well-defined avenue of approach, such as a road or a trail. Information derived from this type of configuration presupposes that the access to this configuration is limited in nature (i.e. access is usually initiated from the ends of the array, proceeding along the entire length of the array. The sensors are deployed so as to take maximum advantage of that characteristic. An area

array, on the other hand, is usually employed to cover a given general area. Unlike the trail array, the sensors of this configuration must be positioned to provide maximum coverage, as opposed to maximum utility. In both types of arrangements, mixes of sensors are normally employed to provide the maximum amount of information possible. Sensors can also be roughly divided into two classes: detection-only type sensors, which are normally utilized to detect a target's presence (but are not normally used to classify a target), and classification sensors, which are normally used to classify the target. As noted before, a mix of both types of sensors is normally found [6].

4. Advantages of Remote Sensor Employment

The employment of remote sensor surveillance systems provide some unique advantages as compared to other methods of intelligence collection [1]. Among the advantages are:

a. Timeliness

Remote sensors can be employed to acquire information in a timely manner; i.e. real time or near real time.

b. All Weather

Remote sensors are significantly less weather restricted than most other means of information collection.

c. Full Time

The majority of remote sensor devices operate without regard to conditions of fatigue.

d. Expendability

All remote sensor devices are expendable and may be employed in high risk environments.

e. Survivability

Areas in which remote sensor devices are employed may be attacked by supporting arms with a high degree of probability that the remote sensor devices will survive undamaged.

f. Covertness

Properly emplaced, employed, and concealed, remote sensor devices can remain undetected.

g. Flexibility

Remote sensor devices may be employed in a variety of situations or in a variety of configurations, and may be emplaced by a variety of means.

5. Disadvantages of Remote Sensor Employment

The employment of remote sensor surveillance devices has its disadvantages as well [1]. The most notable disadvantages are:

a. High Cost

Most remote sensor devices, because of the desired collection capabilities and the required ruggedness built in for field use are relatively high cost items.

b. Limited Discrimination

Present generation remote sensor devices have limited or no ability to react to more than one type of activation stimulus.

c. Limited Life

Remote sensor devices are dependent upon battery power, which limits the useful lifespan of the device. The lifespan is further limited by activation rate and activity rate.

d. Jamming

Present generation remote sensor devices which report by radio link are susceptable to active interference.

e. Range

The detection range of most present generation remote sensor devices is quite limited.

f. Location

Given the limited range of most devices, the effective utilization of the devices requires accurate information about its location.

g. Enemy Reaction

Remote sensor devices are subject to destruction, removal, or boobytrapping by enemy forces.

6. Factors for Consideration in the Employment of Ground Sensor Devices

The effective use of remote sensor devices in a tactical environment depends largely upon a thorough knowledge of the current tactical situation and of planning considerations which apply to the intended area of operations [1,5,7]. Some of the factors to be considered in the employment of remote sensor devices are:

a. Mission of the Unit Being Supported

The task of the unit to be supported by ground sensors has a direct bearing on the number and type of sensors used, and the method in which they will be employed. As an example, if the unit to be supported is conducting an attack, then the ground sensors in support will likely be employed to give that unit commander as much information as possible about enemy troop movements in that immediate area, especially along likely avenues of approach.

b. Type of Coverage Desired

The type of coverage desired will determine the number and types of sensors used. If, for example, blanket coverage of a particular area is desired, many more sensors of all types will be required.

c. Terrain Aspects

Remote sensors are affected to varying degrees by terrain factors. Some of the factors to be considered are

soil type and composition, vegetation (size, type, and density), limitations of topography, road and trail networks, waterway networks and associated tide levels, ambient seismic and acoustic background noise levels, and indigenous population density. These factors will affect the reliability of different classes of sensor output to varying degrees.

d. Weather Considerations

Remote sensors are affected to varying degrees by weather factors. The following factors are considered pertinent to the planning for use of ground sensors: precipitation (type, quantity, duration, frequency, and effect on soil compaction), wind velocity and direction, humidity, and cloud cover.

e. Enemy Considerations

Factors about the enemy to be considered include activity, intensity, and method of conflict.

f. Support Requirements

Among the support factors to be considered for sensor operations are operator proficiency, equipment and personnel available, and methods of delivery.

7. Methods of Employment

The delivery of remote sensors can generally be divided into two broad catagories: air and ground delivery. A third method, delivery by artillery, is currently being

developed and will be an available option when a new generation of ground sensors is fielded. Air delivery may be accomplished by means of selected fixed or rotary winged aircraft in the Army, Navy, and Marine Corps inventory. Aerial delivery is usually the most feasible means of delivery for sensors designated for type III employment, or for type II employment sensors in areas that are heavily dominated by enemy forces [5]. The primary disadvantage of aerial delivery is the resulting loss of accuracy than occurs when sensors are dropped from vehicles that are moving at a moderate to high rate of speed. In contrast, hand emplacement is generally considered to be more accurate, and is usually accomplished by the team that is directly controlling the sensor assets [1]. The primary disadvantages of hand emplacement are the weight limitations imposed by the manual mode of transport, the susceptibility to enemy action on the part of the emplacement team, and the lack of range that an emplacement team enjoys.

8. Remote Ground Sensor Equipment

As was mentioned previously, the present generation of ground sensor equipment was originally designed for use in Southeast Asia. Presently, a full allowance of phase III equipment (the direct product of the modifications made to the original equipment design) has been provided to each of the three active Marine Corps divisions [1]. For a full

listing of the equipment presently available, refer to ref. 4.

In the following discussion only that equipment which pertained to this thesis effort is discussed. For a detailed description of the equipment currently in the Marine Corps inventory, consult ref. 4. A description of the equipment can be broken down into two areas: sensor devices that were actually considered, and recording equipment. All other component items (radio relays, antennaes, and the like) had no bearing on the thesis effort and hence were ignored.

The piece of recording equipment that was of interest to this thesis effort was the signal data recorder. Actually, this piece of equipment had no direct impact on the thesis itself, but as it presently serves as the means of interface between the sensors, and the operator, it was felt that a brief study of this device would assist in the understanding of how the present system functioned, which in turn might provide some clues as to how to approach the problem. The recorder is actually a simple plotting device, an X/T plotter. Information (i.e. activation signals) are represented graphically in one dimension (the "X" dimension). Time histories are kept of the activations as the paper moves by the the graphic pens of the device at a fixed rate of speed (the "T" dimension). Activations are thus recorded as "tick" marks, and are time ordered, being displayed graphically as they occur [8]. It is these sets

that the sensor operator interprets. An example of this type of output is given in figure 1.

The only sensor considered for the thesis effort was a MINISID (in the seismic only configuration). The MINISID responds to seismic activations; that is, it detects vibrations in the earth that occur when a person or vehicle causes seismic activations within its detection radius. When a person or a vehicle moves across the ground, vibrations are caused by that passage. If a seismic sensor is sensitive enough (i.e. the activation is within the sensor's detection radius) then the sensor will detect the vibrations and transmit a signal indicating that a detection has been accomplished. Once the transmission is sent, the sensor "turns itself off" (inhibits itself) for a fixed length of time. The minisid functions in precisely this manner. Diferent sensors usually have different levels of sensitivity depending on the type of target; the MINISID is no exception. The MINISID has an inhibit time of ten seconds, and is classified as a detection-only sensor. Functionally, the MINISID of capable (on the average) of detecting a person at a distance of thirty meters; for vehicles, the MINISID is capable of detection at a distance of one hundred meters. (This varies considerably with the operating characteristics of the sensor itself, the type of the environment in which the sensor is placed, and the type of vehicle itself - the detection radius for a tank is

normally about five times the above stated distance). A MINISID weighs nine pounds [8], and has a minimum rated lifespan of thirty days (the actual lifespan may be quite a bit longer - it is dependent upon the activation rate of the sensor over time). The MINISID is unique in that it can operate independently (in the "seismic only" mode) or it can act in concert with (attached to) other sensors. This thesis considered the MINISID in the "seismic only" mode.

C. AVAILABLE METHODS OF PROCESSING GROUND SENSOR INFORMATION

Other attempts have been made in the past to mechanize the processing of ground sensor information, or to enhance the operator display to facilitate processing. To date none of the developed methods have been implemented by the Marine Corps; the method of processing ground sensor information is still manual in nature. Several of the other methods are mentioned in passing to give the reader an idea of the scope and the direction of previous efforts. Finally, the present manual method of processing ground sensor information is reviewed.

Probably among the first innovations, TMP (Terminal Message Processor) attempted to enhance the operator display to facilitate processing. Enhancement was obtained by placing a simple logic circuit between the receiver and the plotting device. When the threshold for activations per unit

time was reached or exceeded, a "pen burn" (a solid activation mark on the chart paper) was caused. As long as the activation rate per unit time continued to meet or exceed the preset threshold, the "pen burn" continued. This distinctive marking on the chart paper attracted the operator's attention; the rest of the processing cycle was manual in nature [9].

Other, more sophisticated, attempts used a computational (mathematical) approach to the processing requirement. An example is the ATTAC alogrithm [10], which seemed to be about the most sophisticated method encountered in the author's literature search. ATTAC divides the problem area into functional spaces and uses, to a degree, different computational methods to process each different area. The alogrithm is based in part upon time/distance formulas, in part upon outside information supplied by the user, and in part upon statistical analysis of the historical data base. ATTAC is mentioned as the problem structure of the alogrithm was used as a starting point in the design of the prototype production system.

Manual processing of raw sensor data and of sensor derived information is the present method used by the Marine Corps sensor operators. The method is the same whether the TSC modules operate independently or as a unit. Within each module (which is a van, correctly described as the Battlefield Area Surveillance System [BASS] van) [4],

recording devices (described previously) are located. The number of sensors and the configurations will vary according to the tactical situation, but the sensors are normally grouped logically when attached to the plotter so as to facilitate recognition of target activations when they occur. An operator may redefine this grouping as needed as the method of switching is manual (the sensor receiver leads are physically plugged into the plotter jacks). As information is received, indicated by "tick" marks on the plotter paper, the operator looks for distinctive patterns (generally a stairstep type of arrangement, as activations are usually received in a typical array arrangement in a time-ordered fashion). When a distinctive pattern is recognized, a target track is initiated - the operator calculates the target's velocity, and coupled with any other type of information he may possess at the moment (such as data from a classification sensor) he makes a determination as to the type, number, direction of travel, and speed of the target. This information is recorded on a standard report form, and the information is then passed to the next level in the chain of command [5,6] In this manner, information is input into the intelligence system and the tactical reporting system. For a detailed description of the various processing techniques available to the operator, see ref. 6.

III. DESIGN

A. PREFACE

This chapter is divided into two major sections. The first section discusses the structure and the characteristics of knowledge-based production systems in general; the second section details the design of the prototype production system developed as a result of this thesis effort. In the discussion of the prototype system a definition of the problem is given, the system objectives are described, and the structure of the system is discussed.

B. GENERAL

The discussion of knowledge-based production systems is further broken down into five sub-areas. The first is a discussion of the structure and characteristics of knowledge-based problem solving production systems in general. The second is a discussion of appropiate and inappropiate domains for knowledge-based production systems. The third and fourth sub-areas are a discussion of the advantages and the disadvantages of knowledge-based production systems. The fifth is a discussion of the unique features of a hierarchical production system, which is the particular classification of the prototype system developed.

The first four sub-areas should give the reader an understanding of the structure, characteristics, and areas of employment for knowledge-based production systems; the fifth sub-area should allow the reader to build upon that understanding in the domain specific to the thesis-related problem.

1. Structure and Characteristics of Knowledge-based Problem Solving Production Systems

A problem solving production system may be viewed as consisting of three basic components: a set of rules, a data base, and an interpreter for the rules [3]. A rule is an independent "chunk" of knowledge about the problem space and consists of two parts: a situation recognition part (logically an "if" clause) and an action part (a "then" clause). The situations that trigger rules are specified combinations of facts that exist in the data base. The actions that are performed by the rules are restricted to being assertions of new facts deduced directly from the triggering condition(s) [2]. The recognition part of a rule may be viewed as a passive operation of perception, while the action part may be considered as one or more primitive operations to be performed upon a subset of the data base [3]. In this manner, communications between different rules in the system is both direct and restricted to a single channel - the data base.

The data base is a collection of all the facts known to the system about the world (the environment in which it operates) at a given moment. The interpretation of facts in the data base is dependent in large part upon the nature of the application, but it is important to note that the data base represents the sum total of information known to the system (i.e. the data base is unified). Unlike algorithmic methods or structured languages, in a knowledge-based production system there is no provision for separate storage of control state information. The unified data base is universally accessible to every rule in the system, so that anything put in the data base is potentially detectable by any rule [3]. This has a definite impact upon the control structure of the production system.

The interpreter is the source of control for the knowledge-based production system. In simplest terms, the interpreter may be viewed as a select-execute loop, in which one rule applicable to the current state of the world is chosen and then executed. Its action results in a modified data base, and then the selection process begins again [3]. In the simplest production system this process consists of merely scanning the premise conditions of each rule in turn until a match is found with a subset of the data base, performing the action required by the rule identified, and then starting over. This alternation of selection and execution is an essential element of production system

architecture that is responsible for one of the fundamental characteristics of production systems - the complete dependence upon the state of the data base for control. By choosing a rule for execution on the basis of the total contents of the data base, a complete reevaluation of the control system is being performed at every cycle [3]. This is different from procedurally oriented approaches, in which control flow is dependent upon the process then being executed and upon the "value" of the control variables, which may represent only a very small part of the information available to the system at any given moment. With such an "unstructured" control structure, knowledge-based production systems are potentially sensitive to any change in the state of the world within the scope of a single cycle [3].

The above represents a basis from which any production system may be viewed. Many variations are possible, however; any departure from the "pure" structure defined above is usually in response to the structure or complexity of the problem space. The simplest evaluation of production rules involves a matching of literals (system-specific atomic elements of information, usually represented by a single symbol) on the left hand side (the premise side) of the rule, and replacement of the matched literals by the literal contained in the right hand side (the action side) of the rule. This is a simple example of

forward chaining, whereby a conclusion is drawn from the successive application of rules in a deductive manner; i.e. deductions are formed from a given set of facts. A diagram representation of forward chaining may be found in figure 2.

A variation might be the matching of the right hand side of a rule, with the action to be taken contained on the left hand side. This is an example of backward chaining whereby a hypothesis is formed about the state of the world and the data base examined to determine the degree of support present for the hypothesis. This represents an inductive rather than a deductive process, and is well illustrated in the MYCIN production system [11], which operates in the area of clinical medicine. A diagram representation of backward chaining may be found in figure 2. Another variation is that the premise condition(s) might be a complex set of actions in themselves, such as an extensive searching requirement. This technique is utilized in the HEARSAY production system [12] which operates in the area of speech recognition.

The representation of data (in the simplest case, as a set of literals) and the control structure (in the simplest case, no control structure at all) are also likely candidates for variation. The data representation might be multi-dimensional, as is the case in the SU/X production system [13] or it may consist of complex structures. The order in which rules are evaluated may have a tremendous

impact upon the efficiency of a system, as an ordering scheme may significantly reduce the amount of actual processing required to reevaluate the control state every cycle. This requirement will be reflected in the control structure, which itself may be quite complex. A particular variation is to pass control information explicitly in the data base to cause some predetermined evaluation order of the production rules. Another more flexible means of control is through the use of meta-rules, as is done in MYCIN, where the problem of ordering rule evaluation is viewed as just an extension of the problem space. Rules (meta-rules) that will cause the necessary ordering may then be formulated and applied, thus extending the knowledge of the program in a uniform manner. Carried to the extreme, the technique of implementing higher order rules can be advanced to any level desired. Yet another method of ordering rule evaluation, decomposing the problem space into levels, is later described in detail. Each of the above variations (and others not mentioned), while seemingly better fitting a system to a particular problem space, may in fact induce complications in the operation of that system. This may prove to be a limiting factor in the utility of such a system at a later date, as more information about the problem space is obtained. Nevertheless, the performance and structure of any given knowledge-based production system may still be viewed with respect to the concept of the "pure"

production system embodied in the three basic components described earlier.

2. Appropiate and Inappropiate Domains for Knowledge-based Production Systems

Program designers have found that production systems easily model some problem spaces ("space" is defined to mean the sum of all possible problem states - the sum of all the possible combinations of variables in the problem set); some problem spaces, on the other hand, lead to extremely awkward representations [3]. The first criterion for judging whether a problem domain is adequate for a production system is the amount of information available about the domain. If little or no information is available, a knowledge-based production system will probably be ill-suited, as there is little knowledge to base the production rules upon. Likewise, an area in which knowledge is highly formalized and structured will also probably be ill-suited; an algorithmic technique will probably be more efficient overall. An example of this condition might be in the area of Linear Algebra, where the amount of available information is large and highly structured. A production system to solve problems in Linear Algebra might therefore be ill-suited, especially when compared to the number of algorithmic techniques already available. The second criterion concerns the independence of the knowledge primitives (basic knowledge "chunks") of a

problem space. A problem space in which knowledge about the problem can be represented as the sum of many independent states (a state is defined as the values of the program variables at any given moment) is probably well-suited to production systems, as each rule is an independent, modular "chunk" of knowledge. Division of the problem space information into production rules is therefore probably natural and direct. A good example of this condition is the area of clinical medicine, the domain in which MYCIN operates. The sum total of information, both factual and judgemental, about clinical medicine seems to be largely representable in a large number of basic states of knowledge, each independent of the other. A third distinction might be the complexity of control flow [3]. The control structure of a production system is normally simple, visible, and directed through a single channel - the data base. Problems requiring indirect or complex control structures (complex looping or recursion, for example) are probably ill-suited to a production systems approach.

Signal understanding tasks, of which the problem under consideration in this thesis is a member, seem well suited to the application of knowledge-based production systems when analyzed in view of the distinctions presented above. In many signal understanding areas, the amount of information known about the problem space is large, but is neither complete nor highly structured or formalized in

nature, especially in the task of correlation. Many signal understanding tasks seem to be a composition of many independent states of the problem; an example might be in the area of acoustic signals, where the understanding of the signal is dependent in part upon the characteristics of the signal itself, which in turn is dependent upon the strength, source, propagation characteristics, wavelength, etc. of the signal. These divisions of the problem into its "primitive" elements that may each be considered separately reflects the nature of the problem space - a problem that consists of many independent states. Finally, the nature of the control structure required to sucessfully recognize and correlate signals in specific domains seems to indicate that knowledge-based production systems can be readily applied. Interpretation of a signal in such an area is directly dependent upon the state of the data base at that given moment; the manner in which to proceed, dependent upon the state of analysis of that signal, also then seems to be dependent upon the state of the data base at that moment. Such a requirement for a variable control structure lends itself directly to application of knowledge-based production systems.

3. Advantages of Knowledge-based Production Systems

From the foregoing discussion, several advantages of knowledge-based production systems are immediately apparent.

Representing knowledge as production rules forces a homogeneous representation of knowledge [2]. Because each production rule is independent of the others (at least in theory !), an incremental approach to competency is possible; reasonable behavior is demonstrable in a problem area where knowledge about the problem space is incomplete or inexact in nature. Because of rule independence, rules could even be removed (to a certain point) and the system would still exhibit some sort of "reasonable", although reduced (in the sense of competency) behavior. With the proper representation of knowledge, the intuitive or heuristic knowledge of an expert may also be modeled by the use of production rules. Unlike algorithmic techniques, production systems allow unplanned but useful interactions - a piece of knowledge can be applied where and when it is appropiate [2]. Finally, because production rules are modular "chunks" of knowledge, a deduction represents a chaining of the appropiate productions; explanation of behavior of a knowledge-based production system is thus facilitated.

4. Disadvantages of Knowledge-based Production Systems

Several diadvantages accrue when using the production rule method. It is sometimes difficult to express knowledge (especially informal knowledge) in a predetermined format. This gives rise to the expressing of that information in an

indirect manner, which might be difficult for the user to understand. In a large system, the addition of rules may cause side effects not anticipated by the designer, such as when the insertion of a new rule directly contradicts a rule already in the system. Because of the non-structured manner of the production system, changing a rule may lead to changes in other rules which may lead to still more changes; the cumulative effect may be difficult to control [3]. Finally, reevaluation of the program control state every iteration can be computationally expensive. This may tempt the designer to implement some special control structure not available to every production rule. This is directly contradictory to the spirit of the control structure of a production system, and usually leads to awkward structure modifications at a later date [3].

5. Hierarchical Knowledge-based Production Systems

Hierarchical systems may be viewed in the same context of the three basic components of production systems discussed earlier; the discussion of the problem domain, and the advantages and disadvantages of production systems in general are all equally applicable. The distinctive characteristic of the hierarchical production system is the structure of the system itself. In a non-hierarchical system, all the production rules have access to the entire data base; this is one of the fundamental characteristics of

production rule systems. In a hierarchical system, however, the problem space and resulting data base is divided into distinct levels, usually proceeding from the less abstract to the more abstract. Typically, each production rule may have total access to only that part of the data base which it relates to. Instead of being global in scope, each rule normally concerns itself with only one or two levels of the data base structure. This in itself represents a control structure of sorts as control information is no longer accessible to every rule in the system; the order of processing may in fact be predefined.

Of course, a problem that is defined in terms of a hierarchical production system must be representable in that form - as a series of discrete levels of information and control. The advantage of a hierarchical structure lies in the fact that in a non-hierarchical system, the number of rules may grow so large so as to effectively prohibit the reevaluation of the control state every cycle within given problem constraints, whereas in a hierarchical production system the tendency toward this characteristic is eliminated to a large degree. Actually, a hierarchical system may be thought of as a series of non-hierarchical systems, the output of a lower level becoming the input of the next higher level. Viewed in this manner, it is easy to see that the basic tenets of general knowledge-based production systems are directly applicable.

A good example of hierarchical production system is the HEARSAY system [12], which operates in the area of speech recognition. Rather than consider the entire domain of the problem space on a single level (thus making the data base universally available to all the production rules in the system), HEARSAY divides the problem of speech understanding into six distinct levels, each level of which seems to represent a natural level of interpretation of the problem space. In the area of speech recognition, these levels are the parameter, the segment, the syllable, the word, the word-sequence, and the phrase [12]. A parameter is that digitized acoustic signal which is received in a ten millisecond time slice, and represents the most basic element of information in the HEARSAY system. The next level of information contains segments, which represent hypotheses about what useful information is contained in the parameters. Syllables are formed from segments, words are formed from syllables, word-sequences are formed from words, and phrases are formed from word-sequences, completing a cycle through the program levels. At every level, competing hypotheses are represented, reflecting the heuristic nature of the problem. Viewed as a series of single-level production systems, the output of a given level of the HEARSAY system represents the input to the next level. The information elements of any given level thus represent the atomic elements of information (in which the representation

of the problem may be expressed) at that level. Production rules associated with that level of information need only be concerned with the classification of information with which they are familiar; partitioning of the data base thus has the effect of denying access to parts of the data base for a given level of rules in the HEARSAY system.

C. A PROTOTYPE PRODUCTION SYSTEM TO PROCESS GROUND SENSOR INFORMATION

This section is divided into two major subsections, each concerned with a facet of the design of the prototype system developed as a result of this thesis. The first subsection, the overview, gives a definition of the problem and states the goals of the design effort. The second subsection details the design of the prototype system, comparing and contrasting it with the structure and characteristics of a model hierarchical production system, discussed earlier.

1. Overview

As may be inferred from earlier discussions, the problem space to be considered in a solution to the problem of processing information from ground sensors is potentially immense. Different tactical situations, different types of sensors, and different environments combine to form a problem space that is both highly variable and diverse. The

strategy for solution must therefore be variable and dynamically based; an adaptive capability on the part of the processor is the rule rather than the exception. These factors, however, represent only one facet of the problem at hand. Of possibly more concern is the effect on the solution space that the operator himself has. In the case of present operations, where the processor is human, the possible operator variability is large. Not all operators possess the same degree of expertise; the amount and type of experience and training that human operators possess varies greatly [13]. The intensity of the environment also affects different operators to varying degrees; the effects of fatigue on sensor operators is fairly well understood in a high intensity environment [13] but the tolerance range of individual operators in a given situation can only be surmised. As the amount and quality of the information available from ground sensors is at present dependent upon the human operator, it may be surmised that, even in a "perfect" (i.e. completely known and understood) environment, the utility of such information in the present system is much less than is theoretically possible, simply because of operator variance.

The goal of this effort was to demonstrate an alternate method of processing ground sensor information, which, if developed to its full extent, might yield some tangible benefits in the areas of consistency, flexibility,

and reliability. Since this problem area is similiar in nature to a host of others in the military, it might be inferred that this type of solution would be applicable in other related areas of processing and correlating information. It was realized that time and resource constraints would limit this effort to a small subset of the problem space; the immediate goal was therefore to design a "core" system that would be capable of expansion as required or when possible.

2. Design of the Prototype Knowledge-based Production System

At the outset, it was decided to retain as much conceptual simplicity as possible in the design. While perhaps not the most efficient method of implementation, it was felt that a simple design would promote ease of understanding and allow for further expansion. The decision was made, therefore, that the design would reflect, to the greatest extent possible, the structure and simplicity of a "pure" production system, described earlier.

Several production systems already in existence provided clues to the direction that the design effort should possibly take. The two most influential systems already in existence were the HEARSAY system [12] and SU/X [14], both of which are hierarchical knowledge-based production systems. As was previously mentioned, HEARSAY

deals in the realm of speech recognition, and is essentially a forward chaining system. SU/X approaches the problem of recognition of continuous signals, and employs both forward and backward chaining. The control structure of both systems is fairly complex.

The ATTAC algorithm, mentioned previously, provided clues as to the structure of the problem space; in fact, the designers of the algorithm provided a structure of the problem space that was useful as a starting point for the design of the prototype production system. A number of the characteristics of the problem, some already mentioned, provided constraints to the design in one form or another and are listed below:

a. The problem space was potentially a large search space.

b. Potentially diverse sources of knowledge, not only from the sensors, but also from the operator, knowledge about the environment, knowledge about sensor characteristics, and knowledge from outside agencies and historical records tended to complicate the representation of knowledge.

c. Error and variability. The possible noise that could corrupt system input has already been alluded to.

d. An experimental approach was needed for system development. It was anticipated that a need for iteration existed as additional information that impacted upon system became available.

e. Performance. For the purpose of the thesis, efficiency in the sense of storage utilization and performance in the sense of the production of information in a timely manner were largely ignored, since the goal was to investigate and demonstrate a concept. It was felt, however, that some sort of reasonableness in terms of amount of computation should be given some consideration.

f. User interface. It was anticipated that during a later stage of development a capability for the user to ask questions about the decisions the system made would be incorporated.

Given the above, the overall design of an integrated production system was initially presented as a thesis proposal, and is presented as figure 3. Figure 4 represents a conceptual diagram of the design of the production system itself in its present configuration.

The concept of the blackboard for the representation of the global data base was borrowed from the HEARSAY architecture directly. The blackboard is partitioned into distinct information levels as depicted in figure 4. The major units on the blackboard are hypotheses, a set of which (at any given level) is an interpretation of a portion of the problem space at that level. Hypotheses, therefore, are the primitive elements appropiate for representing the problem at that given level; each level becomes a different representation of the problem space. It is interesting to note that, while the present representation of the solution space as a set of hypotheses is a powerful mechanism, other advantages also accrue. In the present system configuration, many of the advantages of representing the solution space as sets of hypotheses are just beginning to be realized. For example, it is possible to represent the "truth" of a hypothesis conveniently through the use of certainty factors [11]. This element has just been introduced at the highest

levels that have been developed to represent an "ordering" of hypotheses by truth value. This implies that conflicting hypotheses may exist concurrently on a given level, and that the truth value of each implies some information about the degree of conflict. Since a certainty factor may be adjusted dynamically as more information becomes available, the widest range of solutions is preserved for consideration. Other variations and orderings are possible; further exploitation is left to future design efforts.

The decomposition of the problem space into levels is a natural parallel to the decomposition of the program's knowledge into separate production rules [12]. Currently, the production rules need only concern themselves with one or two levels of information - the level being operated upon, and the level in which the result of the operation is placed. This is not, however, an inherent constraint in the system design. It is possible, and probably will become practical to make multi-level transfers of information as more information about the problem space is obtained. The levels themselves form the hierarchical structure of the problem, and may each conceptually be thought of as separate, independent production systems that pass information between themselves. The previous discussion concerning hierarchical systems is directly applicable to the prototype system.

The production rules of the prototype system adhere

to the general structure of a rule in a "pure" production system. The rules in the prototype system are organized exclusively as premise-action pairs. The premise part of the rule is represented by a boolean combination of conditions to be verified against the information on the blackboard. The action part of the rule is a simple series of actions to be taken should the premise part be validated. Because of the nature of the information in the problem space, however, substantial deviations occur from the simple concept of the verification and replacement of literals. A typical rule structure is more on the order of a HEARSAY rule, where possibly substantial processing is required to evaluate the left hand side of the rule, and not one but several actions may result from the verification of the left hand side. The rules themselves are organized by levels of information on which they operate, and have been kept separate, independent, and anonymous, which should assist in future modifications to the system.

As has been noted, much of the structure of a "pure" production system has been retained in the development of the prototype system, but nowhere is this more evident than in the control structure. In the prototype system, as in a "pure" production system, the rules are unordered within a given rule level. Each rule has equal access to that portion of the blackboard which all rules on that level have access to. The one exception to this design standard which was

imposed on the design is the condition that intralevel rules must have priority to be scanned before any interlevel rules. This was necessary to insure completeness of the information level to be operated on before trying to pass information to another level. This deviation is not regarded as serious since the same effect could have been achieved by including another information level for each set of intralevel rules. Processing in the correct order would have then been assured, but at the cost of adding another, largely duplicate, level of information. Since the two described methods were equivalent computationally, the decision was made to conserve storage resources.

At present, the processing method is strictly bottom up, that is, forward chaining is exclusively employed. Although not an inherent design constraint, the logical flow of the problem to date requires that all processing be completed at a lower level before moving to the next higher level. Currently, this seems to be a natural representation of the problem space; as new information is added, however, this correspondence may change. A change of that type can be incorporated with no modification required to the rule handler.

The rule handler itself simply tests for the level to be executed upon return from a previous level, and directs control to the newly identified level of processing. It is possible, therefore, to cycle through the levels

progressively (as is the case in the present configuration), or to jump between levels both in an upward and downward direction. As a general condition, a rule which places information on a lower level or intralevel returns control to that level, but as was previously mentioned, this is not an inherent design constraint. This condition can easily be amended or reversed, as true control is dependent solely upon the state of the blackboard. Control is thus very flexible, and in accordance with the principles of a "pure" production system.

IV. IMPLEMENTATION

A. GENERAL

The implementation chapter is divided into four major sections. This section discusses the prototype system's general characteristics, the second discusses the structure of the prototype system. The third section presents the capabilities of the system in its present configuration, while the fourth gives an example of one of the production rules of the system (a sample calculation using the sample rule is also presented). This chapter is designed to familiarize the reader with the program structure and functions, and to provide the context in which the detailed code may be analyzed.

The prototype system developed as a result of this thesis was written in the C language; the system source code is contained in its entirety in the detailed code. The program is designed to run under the UNIX operating system with the standard I/O package. The source code takes approximately two hundred fifty-three thousand bytes embodied in five thousand seven hundred lines of code. As evidenced in the detailed code, however, the program is heavily commented which accounts for a large share of the source code size. The object code itself takes approximately sixty-two

thousand bytes. In its present configuration, the system contains 203 separate routines, in 19 separate files. The program as a unit is capable of and does allocate and deallocate storage dynamically so as to make the most efficient use of available main storage.

B. PROGRAM STRUCTURE

The program in its present configuration is structured into five logical modules - the data generator, the support routines, the "core" production system, the program data base, and the generator data base. Each of the modules is detailed below.

The data generator may be logically broken down into two submodules: the target list handler and its associated support routines, and the data generation submodule. The target list handler is responsible for the formation and upkeep of the target list (which contains the active targets for which data may be generated and their characteristics), and of providing information about the target list to the user's terminal or the line printer on user request. In its present configuration, the target list handler is capable of handling up to six different types of targets, coded and classified as follows:

type	classification
0	No target
1	Personnel
2	Wheeled vehicle
3	Tracked vehicle
4	Rotary winged aircraft
5	Fixed winged aircraft
6	Shock wave

The target list itself is stored as a series of five one-dimensional arrays; a particular index serves to identify that particular target's type, velocity in the X direction, velocity in the Y direction, location in the X direction, and location in the Y direction. At the present time, the target list handler is capable of handling up to ten different targets simultaneously. At present, the user may add or delete a target, change a target's parameters, print the target list at his terminal, or write the target list to the line printer. The target list handler is user friendly in the sense that it checks for common or boundary errors, reports the error to the user at his terminal if one occurred, and reprompts the user for the needed information.

The data generation submodule is responsible for generating the sensor data that would be produced if actual sensors were responding to the targets in the target list in real time. The generation submodule is also responsible for writing the data generated to a permanent file for later

call and use by the executive routine of the main program. The data is generated by comparing target and sensor locations, and the sensor activation radius for that particular type of target. If the target is within range of the sensor and the sensor is able to activate, then an activation is generated for the sensor. Activation data is stored as a series of bytes, one set of bytes per sensor track (a track is defined as a channel required exclusively by a sensor to report information). Each byte of data contains six bits of information; the first bit of a data byte is used for control purposes, and the last bit of the byte is always set at zero to prevent sign extension for storage and control purposes. The number of bytes (a set) per sensor track is dependent upon the time window (the length of the simulation period) and the number of iterations per unit time. At present, each byte of information represents six-tenths of a second; the simulation of a simulation period is thirty seconds.

The data generator at present is a "pure" generator in that any target activations that are allowable are faithfully recorded. This is in contrast with real world data generation in that real world generation would be complicated by random noise, uneven sensor detection radii because of soil and atmospheric conditions, etc. "Pure" data generation thus simplifies the problem; this was considered appropiate at this point in the effort. The present

configuration generates data for the MINISID (in the seismic only mode) type of sensor only (the sensor was previously identified in Chapter II), but the structure of the generator makes it easily expandable when it becomes desirable to add other types or configurations of sensors. As a final note, the data generator contains a linked list structure, the use of which is hidden from the user. The nodes of this structure contain additional information about the sensor tracks and link the sensor to its particular data track(s). The routines of the data generator are grouped into a single file to permit easy removal should actual input from real ground sensors become available.

The support routines perform heavily used primitive functions (such as getting a one character, a floating point, or an integer response from the user) and as such may easily be modified to suit the medium of the response. The support routines also include list primitive functions for the linked lists used in the program, and user query routines for inputting information into the program. The list primitive functions consist primarily of insert and delete routines for a node for every type of linked list used by the program (there are currently twenty-two different types of linked lists), and write-to-printer routines for every type of linked list. The write-to-printer routines allow the system developer to dump lists as required for program debugging and validity checking, and

are available each program cycle. The final class of support routines handle user queries, give the user the ability to add information into the system, change information already present, or delete information from the system. At present, the only routines actually implemented allow the user to add sensors into the system. These routines establish the conditions necessary to augment the system structure with the new sensor automatically, or prompt the user for the information required to establish the necessary links. Like the data generator, these routines are user friendly.

The core production rule system is the heart of the prototype system, and consists of the executive routine, the rule handler, and the production rules themselves. The executive routine operates in a cyclical manner, and is responsible for controlling all of the other routines of the program in a direct or an indirect manner. The major responsibilities of the executive routine are to cause a response to user queries or to cause acceptance of user input or requests, to cause the sensor data to be generated, to cause the sensor data to be read in from the data file, and to cause the data to be processed by the rule handler. If the user inputs the stopping condition into the program, then the executive routine causes the necessary cleanup and termination routines to occur before stopping program execution.

The rule handler processes the sensor information by invoking the various levels of production rules in the order required. The rule handler directs control to the production rule level of the indicator it receives; the level to be processed is thus dependent upon the state of the production rules which in turn are dependent upon the state of the blackboard at any given moment. In this manner, control is decentralized, and the spirit of the control structure of a knowledge-based production system (i.e. no explicit control structure at all) is preserved. After the highest production rule level has been invoked (i.e. that group of production rules which produce information on the highest level of the blackboard) the level handler returns control to the executive routine.

Production rules are organized by the level of information that they operate on (for example, premise information for production rules that is found on level three classify these rules as level three production rules). Each production rule is a separate routine, and as such, is completely independent of all of the other production rules. The premise part of each production rule is executed in its turn, and if found to be true, the action part of the rule is executed. Rules are executed in the sequence that they appear in a level, but this ordering is not at all important except as already noted in Chapter II.

Special control structures that would order the manner in which production rules are scanned have not been implemented.

The global data base contains the data structures and the program variables that comprise the blackboard. With the exception of program constants and declared external variables (such as the system clock), all blackboard information is represented as linked lists, and gives rise to the structure detailed in figure 4. The types of nodes that comprise these lists and the information that they contain are detailed in the program code. It is important to note that some of the lists are level-specific; these lists represent the particular levels of information depicted in figure 4, and are generally time ordered in arrangement. Other lists, such as those that contain information about individual sensors, sensor arrays, and sensor fields are "common" (shared) by all the levels of information and may be thought of as being included on every level's "accessible" blackboard (that part of the blackboard which is able to be accessed by a particular level of production rules). Many of the program constants establish program constraints and operating characteristics; for example, NPERMAP, a program constant defined on line 34 of the file pgmvar.c in the program code, defines the maximum number of units per bit map, which by default is the maximum number of sensors allowed per array, arrays per field, and so forth.

This arrangement allows easy access to important program parameters, should change be desired.

The generator data base contains program variables and data structures peculiar to the data generator. These are grouped together so as to permit easy removal should the data generator be removed. Any routines that reference data generator variables in the main program are specially marked by bracketing lines of a slash followed by 20 asterisks followed by another slash (such as the "generate data" call in the main program on line 22 of the file grnsensr.c in the program code) for easy identification.

C. PROGRAM CAPABILITIES

The major program capabilities or program operating constraints (some of which have already been described) are summarized below:

1. The data generator is capable of accepting and acting upon a maximum of ten targets simultaneously.

2. The program recognizes six distinct target types (plus a classification of "no target" - type 0).

3. A "time window", or span of time that a given sensor data set spans is currently set at thirty seconds.

4. The maximum allowable number of sensors per array, arrays per field, and fields is sixteen.

5. The rule base is currently structured to evaluate sensor data only from sensors in a trail array configuration. Input will be accepted from sensors located in an area array, but will not be processed.

6. The data generator is presently capable of generating (and the program is capable of processing) data from the MINISID (in the seismic only configuration) type of sensor.

7. The user is presently able to add or delete targets, change target parameters,or print the target list if he so desires every processing period.

8. The user is capable of writing any linked list to the line printer, inputting sensors into the system configuration, or setting the program stopping condition if he so desires every processing period.

9. The program is capable of generating up to and including applicable classification hypotheses (information level 5) for each event hypothesis (information level 4), less target count, target direction, and target length (implementation of these routines are largely mechanical and are considered a simple extension to the program's present capability). These classifications are based solely upon the types of sensors "available" to the system in its present configuration. length (these functions are largely mechanical in nature and are considered simple extensions to the present program capability). These classifications are based

D. SAMPLE RUIE AND COMPUTATION

An example rule is given below to give the reader a feel for the type and structure of the production rules included in the program (the rule is typical of those in the program in the sense of structure and function). A complete listing of the rules in their english language and their coded forms may be found in Appendix A and the detailed code, respectively. An example computation, utilizing the example rule, is presented as figures 5 and 6 to allow the reader to observe the manner in which the program (specifically the rules) functions. Where necessary, required system

conditions and data structures are considered to have been already been established for the purpose of this example. The structures of the nodes in the example, although simplified, are similiar to the structures of the actual program nodes.

For the example presented, a "standard" situation was established for presentation. The standard situation consists of:

1. A trail array consisting of three MINISID sensors (in the seismic only configuration) deployed along a trail.

2. Equal spacing of five hundred meters between sensors.

3. The standard system parameters assigned to the sensors (ten second inhibit time, a detection radius of five hundred meters for the given type target, a maximum expected velocity for the target given of seventeen meters/second, a minimum expected velocity of two meters/second, an average expected velocity of nine point five meters/second, and standard activation settings - active, not hypersensitive)

4. A single target, type tracked vehicle, initially located one hundred meters beyond the first sensor's detection radius, and travelling at the constant speed of ten meters/second parallel to (over) the sensor string.

The sample rule, listed below, is a production rule of the prototype system that operates on information on level two, and puts the result of its processing on level three. A description of the rule is as follows:

1213r1:

IF a sensor is a Class III detection-only type sensor AND if it has filtered data in this time period, the sum of which may extend beyond this time period, AND a "valid set" does not exist for this set of filtered data
THEN define a "valid set" for the sensor, insert the new valid set in the sensor's valid set list,

calculate the time of the centroid of the valid set, mark the valid set "incomplete", set the valid set update time to the present time, and set the valid set type to "new"

In the sample calculation, at time equal to thirty seconds (which is the length of the simulation period for data generation, and by default is the processing interval of the rule set), the target previously described in the "standard" test situation has moved well within the detection radius of sensor one, but is still out of the detection radii of sensors two and three. The target in fact moved within range of sensor one and was detectable at time equal to ten. An activation was therefore caused at time equal to ten. Since the sensors have an inhibit time of ten seconds, sensor one is not able to activate again until tie equals twenty, which it does, as the target is still within its detection radius. Similiarly, an activation for sensor one is caused at time equals thirty. This is the state of the data base known to rule 1213r1 when its premise conditions are scanned, and is reflected in figure 5. The premise conditions are found to be true, and the action part of the rule is executed. The result is shown in figure 6.

V. DISCUSSION OF RESULTS

A. OVERVIEW

This chapter is broken down into two sections: the section concerning testing of the system, and the section concerning the results obtained because of that testing. In the section concerning system testing, the method of testing is described, and test situations utilized to performance test the system are described. In the results section, the results of the performance tests conducted are detailed.

B. TESTING OF THE PROTOTYPE PRODUCTION SYSTEM

Testing of the prototype knowledge-based production system may be logically separated into two distinct areas: validation testing, and performance testing.

1. Validation Testing

Validation testing concerned the testing of all the logical components of the prototype system to verify that they performed as expected (i.e. that not only were results produced in a controlled situation, but that the results were the ones anticipated). This classification of testing was essentially exhaustive in nature - each main data path in each module was tested to insure that the results

obtained by exercising that data path were the results expected. Testing was conducted in a bottom-up fashion, starting with the primitive routines of the system and proceeding through the system support routines, the program modules themselves, the interface between modules, and finally the system itself from the lowest level of processing to the highest. Testing was conducted in an iterative manner - when testing of a given level revealed a structural flaw in a lower level, testing reverted to that lower level upon correction of the flaw. In this manner, undesirable side effects caused by making changes at lower program levels without investigating the effects of those changes were eliminated. As may be surmised, this testing effort consumed much of the time originally alloted for overall testing of the system. As a result, the degree of system development and performance were significantly reduced. A beneficial side effect is that there is a fair degree of confidence in the limited results produced by performance testing; it is felt that the results of the performance testing actually conducted reflect the present state of system development and expertise.

2. Performance Testing

Performance testing was conducted through the use of "standard" sensor scenarios. These scenarios were provided by a Marine Corps officer here at the Naval Postgraduate

School, Captain Charles Dublin. Captain Dublin had previously been connected with the Marine Corps ground sensor program in the capacity of a SCAMP platoon commander, and was familiar with sensor operations. Captain Dublin, provided with the prototype system's knowledge capabilities and system limitations, detailed two "standard" scenarios that would exercise the program's capabilities, and perhaps reflect the program's limitations. These two scenarios may be roughly divided into two different classes of problems: a single target scenario, involving one sensor trail array, and a multiple target, single type, single direction scenario. These scenarios or their derivatives were used as a basis for all performance testing conducted, and are presented as follows:

a. Scenario Number One

Scenario number one consists of a single trail array composed exclusively of five MINISID sensors in the seismic only configuration, spaced at unequal intervals (later selected to be four hundred, five hundred fifty, three hundred, and seven hundred meters, respectively) to provide overlapping and non-overlapping detection radius coverage. Planning detection radii (meters), minimum, average, and maximum target velocities (meters/second) were left at the system default conditions for the system, and are presented, by type, as follows:

Target type	min vel	avg vel	max vel	det rad
Personnel	.5	1.5	3.0	30.0
Wheel veh	2.0	14.0	26.0	100.0
Track veh	2.0	9.5	17.0	500.0
R/W A/C	20.0	60.0	100.0	550.0
F/W A/C	90.0	170.0	250.0	700.0
Shk wave	350.0	350.0	350.0	950.0

The inhibit time for each sensor was the system default (ten seconds for all sensors). All sensors were considered active and functioning normally. The target chosen was a tracked vehicle with a constant speed of ten meters/second, initially placed one hundred meters beyond the detection radius of the first sensor it was to encounter. The direction of travel for the target was parallel (over) the sensor string, passing through the detection radius of each sensor in its turn.

b. Scenario Number Two

The sensor array configuration, the sensor spacing, and the sensor operational characteristics are identical to those presented in Scenario number one. The target, however, was changed to a column of four wheeled vehicles, all travelling at a constant speed of thirteen meters/second. As in Scenario number one, the direction of travel is parallel to (over) the sensor array. All vehicles are equally spaced initially; a distance of one hundred fifty meters is maintained between vehicles. As in Scenario

number one, the lead vehicle is initially placed one hundred meters beyond the detection radius of the first sensor encountered.

C. PERFORMANCE TEST RESULTS

The test of the system using Scenario number one revealed that the system correctly identified the target as a tracked vehicle with a certainty factor of .8076, and a final recorded velocity of 10.02 meters/second. An alternate classification, that of a wheeled vehicle with a certainty factor of .1923 was also established by the system. A detailed time period by time period analysis of system hypotheses revealed that the system had early on considered as many as three target classification types, but as more information became available to the system (in the form of input from all the sensors in the array and from more complete activation sets from each sensor) the system narrowed its choices of classification to the two final target types described, and adjusted the certainty factors accordingly with the increasing amount of information.

The results of the multiple target scenario were not as clearcut. The system correctly identified the type of target with a certainty factor of .725 and a velocity of 13.01 meters/second, but only identified one target instead of the four actually present. The other classification produced was

an alternate (competing) classification; it was that of a tracked vehicle with a certainty factor of .275. Upon closer inspection of the system intermediate operations and classifications, it was discovered that the combination of target speed and separation distance produced overlap - that is, a trailing vehicle would enter a sensor's detection range before the preceding vehicle had a chance to exit that same sensor's detection range. This resulted in spillage of one alarm set into another, a condition that the system in its present configuration is not equpped to handle. In this case, lacking information to the contrary, the system had no choice other than to consider each sensor activation set as continuous, and hence resulting from one target.

D. CONCURRENT TESTING

The above two tests were by no means the only tests conducted on the system; they are presented as a representative sample of the type of tests conducted and the nature of the results obtained. One objective of the thesis effort, already mentioned, was to demonstrate that knowledge could be added to the system in an incremental manner. This was accomplished in part by the modification or replacement of several of the system's rules to reflect a small increase in the knowledge base of the system. A representative example is the elimination of rule 1112r0 and its

replacement by 1112r1, both of which are detailed in Appendices A and B in their english language and coded forms, respectively. In the original rule base of the system, of which 1112rØ is a part, all sensor activations are considered to be valid as received - vagaries in sensor performance characteristics are ignored. In this manner, all sensor activations are received and passed directly into the system data base for further processing; the sensor status remains unaffected whether a sensor is performing properly or not. Sensors do behave erratically on occasion, however; if an erratic sensor is allowed to influence an ongoing classification then a skewed or an erroneous classification will result. The replacement of 1112rØ by 1112r1 was designed to add some additional information into the system concerning the detection of a sensor in a hypersensitive condition (defined as the condition that results when a sensor becomes hyperactive and transmits activations whether a target is present or not). With the additional information, the system successfully identified and eliminated from consideration a sensor that was forced to become hypersensitive (through the simple mechanism of placing a target with zero velocity directly on top of the sensor). This represented a previous improvement over a previous test in which the system included the malfunctioning sensor in the classification calculations, thus generating a false classification. It is sufficient to

note that, although this represents an improvement in system knowledge in that area, it by no means represents perfect information in that area. A sensor that chose to malfunction in the middle of a classification formation invariably skewed the result. More complete information, perhaps causing any classification to be redone when the condition is finally identified, seems to be a reasonable approach.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

From the limited testing conducted, and the limited nature of the subset of the problem space that was chosen for investigation, it was difficult to draw any sweeping conclusions concerning the applicability of a knowledge-based production to this class of problems. The fact remains, however, that in the limited time available and with limited resources, a production system that performed reasonably over a small subset of the problem area was developed. It appears to be feasible to approach the problem of analyzing and correlating ground sensor information through the utilization of a knowledge-based production system. No claim is made concerning the optimality or even continued feasibility of representing the data space of the ground sensor problem in the manner described in this thesis; it is recognized that this attempt at solution may bear little resemblance to an operational production system, should one ever be implemented in this particular area. Given all the caveats above, however, it seems reasonable to assert that knowledge-based production systems offer a possible means of solution to this problem area that ought to be explored.

Even in such a limited effort, the advantages and the potential of a knowledge-based production system were apparent. The potential adaptability of the control structure allowed a tremendous flexibility in the prototype design, even at the present level of development. Circumstances can already be forseen where multilevel transfer of information will be desirable (as in the addition of the capability to process information from classification types of sensors, which should provide direct confirming classification information from basic data level input) thus bypassing several intermediate levels of processing. A change of that magnitude in a corresponding algorithmic technique would potentially be difficult and might even force in a complete redefinition of the problem or a major program restructuring. In the prototype production system developed, however, no change at all is required in the control structure to allow for the processing of that new form of information. A unified data base, properly represented, will allow the user to eventually query the system as to the reasoning for system deductions made, and will allow the system to respond in an understandable and a useful manner. The global availability of the data base also makes introduction of meta-rules possible to establish a uniform control structure to establish the order in which production rules on any given level are considered for evaluation. Representing knowledge

about the problem space in the form of production rules allowed an incremental approach to the level of competence achieved by the system. It became apparent early on that representing information as independent, modular "chunks" of knowledge facilitated the initial establishment and testing of, and the later augmentation of, system knowledge as presently embodied in the prototype system.

The last, and perhaps the most important advantage of all accrued by structuring the problem in terms of a knowledge-based production system was the ability of the system to conveniently represent and process competing hypotheses simultaneously. In the present system configuration, with its relatively simple classification scheme (based solely on the difference between the measured average velocity at a given time and the the median expected velocity for a given class of target), this ability is not critically important. As knowledge is added to the system, however, both in differing form and in increased scope, this capability will almost certainly assume a greater importance. Simply expressed, different hypotheses at any given level of program knowledge representation expresses the sum total of what the system knows about the problem state at any given moment. Since the system is sensitive to any change in the state of the world within the scope of a single cycle, and since that change will be immediately reflected in a change to one or more hypotheses, the user is

potentially kept informed of even the slightest change in the environment in which the system is operating. It must be noted that any other representation of system information has the potential of hiding important information from the user. The representation of the system state as a series of competing hypotheses is thus a very powerful means of the representation of system information.

The disadvantages and uncertainties of the prototype production system developed must also be reported, however, as they reflect upon the applicability, "cost", and the limitations of knowledge-based production systems as they apply to this type of problem area. As indicated by the figures given previously, the "cost" in terms of storage utilization for the prototype system is relatively large. Given the small number of production rules supported at present, this cost at first seems inordinate. Allusion has already been made to the amount of duplicity that is found in the coded production rules; a price is certainly paid to guarantee rule independence. This disadvantage should be tempered by the realization that the choice of language made to implement the system was definitely not an optimal one, and in fact may contribute to this problem to a significant degree. As the system design progressed, it became apparent that the majority of primitive operations that the system performed were of the list processing nature. A list processing language (a good example of a language

specifically designed to process lists is LISP) would probably have represented a better choice of an implementation language. Nevertheless, one of the disadvantages to keeping information chunks independent is that some rules are duplicative in nature, and if the size of the rules are large in general, the storage cost will be high.

The most serious disadvantage, it was felt, was the lack of a degree of confidence that the representation of knowledge (as represented by the form of the data base and the production rules) was adequate in nature. The representation of knowledge is diverse and hotly debated topic, there is very little in the way of guidelines on how to adequately represent knowledge in any domain. There is no assurance that any further advancement in the prototype production system developed will not lead to disaster in terms of representing the classes of information that must be considered to adequately represent the problem space. This lack of assurance is common to an algorithmic approach also, perhaps even to a much greater degree, since one small change has the potential of collapsing the entire control structure of such a system. The realization that "adequacy" of the structure of information representation is not demonstrable in any known sense (even given the inherent flexibility of the control structure) is somewhat disquieting. At present, it cannot be demonstrated that the

approach presented in this design effort offers the necessary flexibility to overcome major conceptual changes.

B. RECOMMENDATIONS

In regards to the prototype system developed, it seems desirable to continue the development effort. Although a new generation of sensors is being developed [REMBASS], and various techniques are being explored that classify targets at great ranges [NOSC], the basic problem of correlation of that information, once recieved from whatever means, in a useful and timely manner remains. Certainly, further development and expansion of the prototype system developed as a result of this thesis is encouraged. Much information about the problem space was derived from this effort to implement a simple system. Much of the processing required on the lower levels of the problem, which at first seemed uncomplicated in nature, actually required some rather complex primitive actions to be taken. Continued development of the system presented will almost certainly produce further insight as to the nature of the problem space, and the knowledge and processing required to represent and produce solutions in the problem space.

As may be inferred from the foregoing discussion, it is felt that the application of knowledge-based production systems offer distinct advantages in this and other related

problem areas. The inherent flexibility of the control system and the independent nature of rules in the rule base lend themselves to the incremental approach to a solution of this class of problem. Certainly, as more (in terms of quantity) and diverse (in terms of potential sources) amounts and types of information become available to the military command structure at every level, the requirement for a highly adaptive control structure to process and correlate that information received will only increase, while the time allotted for the system to perform those tasks (whether the system be composed of machines or human beings) will surely decrease as faster response times are demanded. Because more will be demanded, more will be expected, and in an increasingly shorter time frame. It therefore behooves those who allocate the resources for this capacity to allocate the available resources in an optimal fashion. Knowledge-based production systems seem to offer a viable method of meeting this challenge, especially as processing hardware becomes less expensive, while human resources become more expensive. Further research and investigation into this area of research is therefore certainly warranted.

APPENDIX A - SYSTEM PRODUCTION RULES

RULE NO.	RULE
1Ø11rØ	IF a sensor is a Class III detection-only type sensor AND if its activation value is greater than a fixed upper threshold limit AND if its activation value is greater than the average of the activation values for all the active, non-hypersensitive class III detection-only type sensors in the same sensor array AND the sensor has been non-hypersensitive for the required period of time THEN set the sensor to hypersensitive
1Ø11r1	IF a sensor is a class III detection-only type sensor AND if that sensor is hypersensitive AND if its activation value is not greater than a fixed upper threshold limit OR if its activation value is not greater than the average of the activation values for all the active, non-hypersensitive class III detection-only type sensors in the same sensor array AND the sensor has been hypersensitive for the required period of time THEN set the sensor to non-hypersensitive
1Ø11r2	IF a sensor is a class III detection-only type sensor AND if the sensor is inactive AND the sensor has activation(s) for this time period THEN set the sensor to active
1Ø11r3	IF a sensor is a class III detection-only type sensor AND if its activation value is less than a fixed lower threshold limit AND if its activation value is less than the the average of the activation values for all the active, non-hypersensitive class III detection-only type sensors in the same sensor array AND the sensor has been active for the required period of time THEN set the sensor to inactive

1112r0 IF a sensor is a class III detection-only type sensor AND if it has activation(s) in this time period
THEN create a filtered data set for the sensor (type Ø filtering)

1112r1 IF a sensor is a class III detection-only type sensor AND if it has activation(s) in this time period AND if the sensor is not hypersensitive
THEN create a filtered data set for the sensor (type 1 filtering)

1213rØ IF a sensor is a class III detection-only type sensor AND if it has filtered data in this time period, the sum of which does not extend to the end of the time period, AND a "valid set" does not exist for this set of filtered data
THEN define a "valid set" for the sensor, insert the new valid set in the sensor's valid set list, calculate the time of the centroid of the valid set, mark the valid set "complete", and set the valid set type to "new"

1213r1 IF a sensor is a class III detection-only type sensor AND if it has filtered data in this time period, the sum of which may extend beyond this time period, AND a "valid set" does not exist for this set of filtered data
THEN define a "valid set" for the sensor, insert the new valid set in the sensor's valid set list, calculate the time of the centroid of the valid set, mark the valid set "incomplete", and set the valid set type to "new"

1213r2 IF a sensor is a class III detection-only type sensor AND if it has filtered data in this time period, the sum of which does not extend beyond this time period, AND a "valid set" already exists that is meaningful to this set of filtered data
THEN update that "valid set", update the time of the centroid of the valid set, mark the

valid set "complete", and set the valid set type to "extension"

1213r3 IF a sensor is a class III detection-only type sensor AND if it has filtered data in this time period, the sum of which may extend beyond this time period, AND a "valid set" already exists that is meaningful to this set of filtered data
THEN update that "valid set", update the time of the centroid of the valid set, and set the valid set type to "extension"

1313r0 IF the time of update for a valid set is less than the current system time AND the valid set is incomplete OR the valid set is complete and has a type of "new"
THEN mark the valid set "complete", set the update time of the valid set to the base data time, and set the valid set type to NULL

1313r1 IF a valid set from a class III detection-only type sensor has a centroid time that is less than the starting time that would be required for the slowest possible target to travel to the adjacent sensor that is farthest from the sensor AND the valid set is not part of an event hypothesis
THEN remove the valid set from further consideration

1314r0 IF valid sets from Class III detection-only type sensors are present from at least a minimum correlation set for a trail array, and the valid sets are time ordered by relative sensor position
THEN form an event hypothesis and calculate the target velocity list using the various sensor combinations

1414r0 IF a duplicate event hypothesis exists from the same array
THEN eliminate the later duplicate

1414r1 IF an event hypothesis from an array is a subset of another event hypothesis from the same array
THEN remove the subset hypothesis

1414r2 IF an event hypothesis from an array is a subextension of another event hypothesis from the same array
THEN remove the subextension event hypothesis

1415r0 IF an event hypothesis has been formed at the present system time AND if any velocity from its velocity list is greater than or equal to the expected minimum speed for personnel and less than or equal to the expected maximum speed for personnel OR the minimum event velocity is less than the expected minimum speed for personnel and the maximum event velocity is greater than the expected maximum speed for personnel
THEN form a "personnel" classification hypothesis, calculate the certainty factor of all classifications formed from the event to date, and calculate the classification parameters

1415r1 IF an event hypothesis has been formed at the present system time AND if any velocity from its velocity list is greater than or equal to the expected minimum speed for wheeled vehicles and less than or equal to the expected maximum speed for wheeled vehicles OR the minimum event velocity is less than the expected minimum speed for wheeled vehicles and the maximum event velocity is greater than the expected maximum speed for wheeled vehicles
THEN form a "wheeled vehicle" classification hypothesis, calculate the certainty factor of all classifications formed from the event to date, and calculate the classification parameters

1415r2 IF an event hypothesis has been formed at the present system time AND if any velocity from

from the event to date, and calculate the the classification parameters

1415r4 IF an event hypothesis has been formed at the present system time AND if any velocity from its velocity list is greater than or equal to the expected minimum speed for fixed wing aircraft and less than or equal to the expected maximum speed for fixed wing aircraft OR the minimum event velocity is less than the expected minimum speed for fixed wing aircraft and the maximum event velocity is greater than the expected maximum speed for fixed wing aircraft
THEN form a "fixed wing aircraft" classification hypothesis, calculate the certainty factor of all classifications formed from the event to date, and calculate the classification parameters

1415r5 IF an event hypothesis has been formed at the present system time AND if any velocity from its velocity list is greater than or equal to the expected minimum speed for a shock wave
THEN form a "shock wave" classification hypothesis, calculate the certainty factor of all classifications formed from the event to date, and calculate the classification parameters

1515r0 IF a classification hypothesis is a duplicate, subset, or subextension of another classification hypothesis
THEN remove the duplicate, subset, or subextension hypothesis

```
/* This file contains the data structures and program
   variables used in the AI ground sensor program. */

#include <stdio.h>

/* Data definitions */

#define TMWINDW  30        /* number of seconds per time window */
#define NUMT      2        /* min no. of sensors needed for correlation */
#define NMSNCL    1        /* number of sensor classes */
#define NMSTYP    6        /* number of sensor detection types */
#define NMSSTYP   4        /* number of sensor detection subtypes */
#define NDIRSEN   1        /* number of type of directional sensors */
#define INITTM  -10.0      /* initial last activation time for sensors */
#define DEFSOIL   0        /* default soil code */
#define NO        2        /* boundary indicator */
#define YES       1        /* boundary indicator */
#define NPERSEC  10        /* number of iterations per second */
#define NIL      -1        /* nil pointer definition */
#define NONE      0        /* null type definition */
#define NWINDW   10        /* number of windows for validity checks */
#define NTTYP     6        /* number of target types */
#define VALID     1        /* boolean switch */
#define ERROR     0        /* boolean switch */
#define TRUE      1        /* boolean switch */
#define FALSE     0        /* boolean switch */
#define STOP     99        /* boolean switch */
#define MAXSZ    80        /* maximum number of char per line */
#define TRMINAT  -99       /* termination indicator */
#define ON      -1.0       /* on indicator */
#define OFF     -1.0       /* off indicator */
#define SMOOTH    .1       /* smoothing constant */
#define NOT     -1.0       /* not indicator */
#define NPERMAP   16       /* maximum number of units allowed per bit map */
#define MASMEN    .5       /* sensor inhibit time uncertainty factor */
#define NPRUNIT (NPERMAP / 8) /* number of bytes per element */
```

```
#define NBYTES (NPRUNIT * NPERMAP) /* number bytes per bit map */

/* Other external variables */

struct sensid *sidhdr, *sidtl; /* sensor ID list head and tail pointers */
struct arayid *aidhdr, *aidtl; /* array ID list head and tail pointers */
struct fildid *flidhdr, *flidtl; /* field ID list head and tail pointers */
struct dsensid *dsidhdr, *dsidtl; /* deleted sensor ID list hd, tl pntrs */
struct darayid *daidhdr, *daidtl; /* deleted array ID list hd, tl pntrs */
struct dfildid *dfidhdr, *dfidtl; /* deleted field ID list hd, tl pntrs */
struct fldroot *rootptr;          /* pointer to root */
struct evnt *evhdr, *evtl;        /* event hypoth list head, tail pntrs */
struct classh *clhdr, *cltl;      /* class hypoth list head, tail pntrs */
struct filhyp *filhdr, *filtl;    /* filtered hypoth list hd, tl pntrs */
double atof ();           /* external routine declarations */
double sqrt ();
double pow ();
double fabs ();
float qtfloat ();
float getmin ();
char getans ();
FILE *fopen (), *fp;          /* file pointers */

/* Data structures */

  /* Sensor data list node - unfiltered */
struct sdatau {
  float timact;           /* time of the activation */
  struct sdatau *left;    /* pointer to previous node */
  struct sdatau *right;   /* pointer to next node */
  struct sensnod *prnt;   /* pointer to parent sensor node */
};

  /* Sensor data list node - filtered */
struct sdataf {
  float tmact;            /* time of the activation */
```

```
  struct sdataf *lft;      /* pointer to previous node */
  struct sdataf *rght;     /* pointer to next node */
  struct sdatafl *pnt;     /* pointer to parent list node */
};

  /* Filtered sensor data list list node */
struct sdatafl {
  int fdltyp;              /* type filtering */
  struct sdatafl *fdlpr;   /* pointer to previous node */
  struct sdatafl *fdlnx;   /* pointer to next node */
  struct sdataf *dfhead;   /* pointer to filtered data header node */
  struct sdataf *dfsent;   /* pointer to filtered data sentinel node */
  struct sensnod *dlprnt; /* pointer to parent sensor node */
};

  /* Sensor node */
struct sensnod {
  int sbmapno;             /* bit map index */
  int type;                /* sensor  and type code */
  float slocx;             /* 4 digit UTM X coord sensor location */
  float slocy;             /* 4 digit UTM Y coord sensor location */
  float detdir;            /* detection azimuth */
  float detrad[NTTYP];     /* sensor detection radius by type */
  float accangl;           /* directional acceptance angle */
  float tmactiv;           /* time sensor activated */
  float hs;                /* sensor inhibit time */
  float expect;            /* expectation estimate */
  int numcnt[NWINDW];      /* number of activations previously */
  int soil;                /* soil code */
  float hyprsen;           /* sensor hypersensitivity indicator */
  float offon;             /* sensor functionality indicator */
  int numcont;             /* number of continuous activations */
  float eol;               /* projected end of life time */
  struct sensnod *coloc;   /* pointer to colocated sensor, if any */
  struct sdatau *head;     /* pointer to unfiltered data list head */
  struct sdatau *tail;     /* pointer to unfiltered data list tail */
```

```
  struct sdatafl *hd;      /* pointer to fltrd data list node list head */
  struct sdatafl *tl;      /* pointer to fltrd data list node list tail */
  struct vset *hdr;        /* pointer to valid set list head */
  struct vset *sentl;      /* pointer to valid set list tail */
  struct araynod *parent;  /* pointer to parent array node */
  struct sensnod *prev;    /* pointer to previous node */
  struct sensnod *nx;      /* pointer to next node */
  struct sensid *sidback;  /* pointer to ID node */
};

  /* Sensor ID list node */
struct sensid {
  int idno;                /* 4 digit sensor id number */
  struct sensnod *indic;   /* pointer to sensor node */
  struct sensid *nxt;      /* pointer to next list node */
};

  /* Sensor array node */
struct araynod {
  int abmapno;             /* bit map index */
  int typ;                 /* code for type array */
  int numsen;              /* number of sensors in the array */
  float vmin[NTTYP];       /* min velocity for targets by type */
  float vmax[NTTYP];       /* max velocity for targets by type */
  float vav[NTTYP];        /* average velocity for targets by type */
  char sadajc[NBYTES];     /* bit map for sensor adjacency */
  struct sensnod *headr;   /* pointer to sensor list head for array */
  struct sensnod *sntinl;  /* pointer to sensor list tail for array */
  struct araynod *lf;      /* pointer to previous node */
  struct araynod *rt;      /* pointer to next node */
  struct fildnod *parnt;   /* pointer to parent field */
  struct arayid *aidback;  /* pointer to ID node */
};

  /* Sensor array ID list node */
struct arayid {
```

```
  int araynum;              /* 4 digit array id number */
  struct araynod *pntr;     /* pointer to array node */
  struct arayid *next;      /* pointer to next list node */
};

  /* Sensor field node */
struct fildnod {
  int feldtyp;              /* code for type of field */
  int fbmapno;              /* bit map index */
  int numary;               /* number of arrays in the field */
  char aradajc[NBYTES];     /* bit map for array adjacency */
  struct araynod *header;   /* pointer to array list head for field */
  struct araynod *sntnl;    /* pointer to array list tail for field */
  struct fildnod *fprev;    /* pointer to previous node */
  struct fildnod *fnext;    /* pointer to next node */
  struct fldroot *rootpar;/* pointer to root node */
  struct fildid *fidback;   /* pointer to ID node */
};

  /* Sensor field ID list node */
struct fildid {
  int fildnum;              /* 4 digit field id number */
  struct fildnod *ptr;      /* pointer to field node */
  struct fildid *follow;    /* pointer to next list node */
};

  /* Valid set node */
struct vset {
  int numact;               /* number of activations in valid set */
  float centrd;             /* centroid time */
  float timupd;             /* time the set formed or last updated */
  int compl;                /* complete or incomplete indicator */
  struct sensnod *lnk;      /* pointer to sensor node */
  struct sdataf *lst;       /* pointer to filtered activations for set */
  struct vset *prevs;       /* pointer to previous node */
  struct vset *foll;        /* pointer to next node */
```

```
  struct evntl *higher;    /* pointer to event hypothesis list head */
  struct evntl *eltail;    /* pointer to event hypothesis list tail */
};

  /* Event hypothesis list node */
struct evntl {
  struct evnt *link;       /* pointer to linked event */
  struct evntl *prvious;   /* pointer to previous node */
  struct evntl *follw;     /* pointer to next node */
  struct vset *vlower;     /* pointer to parent valid set */
};

  /* Event hypothesis node */
struct evnt {
  int etyp;                /* type event by sensor class, type and mix */
  float velmax;            /* event maximum velocity */
  float velmin;            /* event minimum velocity */
  float velav;             /* event average velocity */
  int numvset;             /* number of vsets in the event */
  int numcl;               /* number of classifications from the event */
  float updtim;            /* time last updated and type of update */
  struct evnt *last;       /* pointer to previous node */
  struct evnt *fll;        /* pointer to next node */
  struct vsetl *lowr;      /* pointer to valid set list head */
  struct vsetl *vstail;    /* pointer to valid set list tail */
  struct chypl *hgher;     /* pointer to class hypothesis list head */
  struct chypl *chtail;    /* pointer to class hypothesis list tail */
  struct velist *vhdr;     /* pointer to velocity list head */
  struct velist *vsent;    /* pointer to velocity list tail */
};

  /* Velocity list node */
struct velist {
  int sens1;               /* first sensor ID */
  int sens2;               /* second sensor ID */
  float vel;               /* velocity from sensor 1 to sensor 2 */
```

```
  struct velist *vlast;   /* pointer to previous node */
  struct velist *vnext;   /* pointer to next node */
  struct evnt *vpar;      /* pointer to parent event */
};

  /* Valid set list node */
struct vsetl {
  struct vset *vslink;    /* pointer to valid set */
  struct vsetl *lprev;    /* pointer to previous node */
  struct vsetl *lfoll;    /* pointer to next node */
  struct evnt *eparnt;    /* pointer to parent event */
};

  /* Classification hypothesis list node */
struct chypl {
  struct classh *hlink;   /* pointer to classification hypothesis */
  struct chypl *cprev;    /* pointer to previous node */
  struct chypl *cnxt;     /* pointer to next node */
  struct evnt *cparnt;    /* pointer to parent event */
};

  /* Classification hypothesis node */
struct classh {
  int ctyp;               /* type  by sensor class, type and mix */
  int class;              /* classification code */
  float cvel;             /* classification velocity */
  float cdir;             /* classification direction */
  float clength;          /* classification length */
  int cnum;               /* classification number of targets */
  int cloc;               /* classification location */
  float ceftim;           /* effective time of calculations */
  float cupd;             /* time formed or last updated */
  float cf;               /* classification type certainty factor */
  struct classh *hprev;   /* pointer to previous node */
  struct classh *hnext;   /* pointer to next node */
  struct eventl *clower;  /* pointer to event list head */
```

```
  struct eventl *chtl;     /* pointer to event list tail */
  struct fhypl *chigher;   /* pointer to filtered class hypoth list head */
  struct fhypl *fhtail;    /* pointer to filtered class hypoth list tail */
};

  /* Event list for a classification hypothesis node */
struct eventl {
  struct evnt *elink;      /* pointer to event hypothesis */
  struct eventl *elprev;   /* pointer to previous node */
  struct eventl *elnext;   /* pointer to next node */
  struct classh *chparnt; /* pointer to parent  hypothesis */
};

  /* Filtered classification hypothesis list node */
struct fhypl {
  struct filhyp *flink;    /* pointer to filtered  hypothesis */
  struct fhypl *flprev;    /* pointer to previous node */
  struct fhypl *flnext;    /* pointer to next node */
  struct classh *clprnt;   /* pointer to parent class hypothesis */
};

  /* Filtered classification hypothesis node */
struct filhyp {
  struct filhyp *fleft;    /* pointer to previous node */
  struct filhyp *fright;   /* pointer to next node */
};

  /* Field root node */
struct fldroot {
  int numfld;              /* number of fields attached to root */
  char fadajc[NBYTES];     /* bit map for field adjacency */
  struct fildnod *fldhdr; /* pointer to field list header */
  struct fildnod *fldtl;   /* pointer to field list sentinel */
};

  /* Deleted sensor ID list node */
```

```
struct dsensid {
  int didno;                /* 4 digit sensor id number */
  struct sensnod *dindic; /* pointer to sensor node */
  struct sensid *dnxt;      /* pointer to next list node */
};

  /* Deleted sensor array ID list node */
struct darayid {
  int daraynum;             /* 4 digit array id number */
  struct araynod *dpntr;  /* pointer to array node */
  struct arayid *dnext;     /* pointer to next list node */
};

  /* Deleted sensor field ID list node */
struct dfildid {
  int dfildnum;             /* 4 digit field id number */
  struct fildnod *dptr;     /* pointer to field node */
  struct fildid *dfollow; /* pointer to next list node */
};
```

```
/* This file contains the data structures and program variables
   used exclusively in the data generator routine. */

/* Data definitions */

#define MAXTGTS      10     /* maximum number of targets at a time */
#define MAXTRKS     100     /* maximum number of target tracks */
#define MAXMAPX  1000.0     /* maximum map coord X direction */
#define MAXMAPY  1000.0     /* maximum map coord Y direction */
#define MINMAPX     0.0     /* minimum map coord X direction */
#define MINMAPY     0.0     /* minimum map coord Y direction */
#define FFAC        0.5     /* sensor inhibit time uncertainty factor */

/* Other external variables */

int tgttype [MAXTGTS];       /* target type array */
float tgtvelx [MAXTGTS];     /* target velocity array X direction */
float tgtvely [MAXTGTS];     /* target velocity array Y direction */
float tgtlocx [MAXTGTS];     /* target location array X direction */
float tgtlocy [MAXTGTS];     /* target location array Y direction */
float bstim;                 /* window base time */
struct tgttrk *thead, *ttail;  /* head and tail pointers */

/* Data structures */

  /* Target track node */
struct tgttrk {
  int sensrno;               /* associated sensor ID number */
  int nactv;                 /* number continuous activations */
  float lstact;              /* last activation time */
  float lstchk;              /* last node confirm time */
  struct tgttrk *tnext;      /* pointer to next list node */
};
```

```
/* This section contains variables that must be declared external */
int cntinue      1;        /* Variable boolean switch */
float systime  0.0;        /* System time */
float datatim  0.0;        /* base time for data input - used for log test */
float bastim   0.0;        /* working value for base time */
int senscnt      0;        /* count of sensor tracks */
int mask[8] = {0001,0002,0004,0010,0020,0040,0100,0200}; /*bit mask */
char dfilnam[] = "TGTDATA"; /* name of target data file */
char pfilnam[] = "HARDCOPY"; /* name of scratch file for hardcopy */
float mintbl[NTTYP] = {.5,2.0,2.0,20.0,90.0,350.0}; /* min vel table */
float maxtbl[NTTYP] = {3.0,26.0,17.0,100.0,250.0,350.0}; /*max vel table */
float avtbl[NTTYP] = {1.5,14.0,9.5,60.0,170.0,350.0}; /* av vel table */
```

```
/* This section contains external declarations for external variables */
extern int cntinue;        /* Variable boolean switch */
extern float systime;        /* System time */
extern float datatim;        /* base time for data input */
extern float bastim;         /* working value for base time */
extern int senscnt;        /* count of sensor tracks */
extern int mask[8]; /*bit mask */
extern char dfilnam[]; /* name of target data file */
extern char pfilnam[]; /* name of scratch file for hardcopy */
extern float mintbl[NTTYP]; /* min vel table */
extern float maxtbl[NTTYP]; /*max vel table */
extern float avtbl[NTTYP]; /* av vel table */
```

```
/* This module is the control and level handler sections for the ground
   sensor AI program.  The control section controls the overall flow of
   the program from start to finish.  The level handler directs control
   to the appropiate program level for action, and returns control to
   the control section indicating the next level to be processed. */

#include <stdio.h>
#include "pgmvar.c"          /* include program data */
#include "genvar.c"          /* include generator data */
#include "exvar.c"              /* include external variables */

main () {
  int indcat, tempr;
  initlz ();                 /* initialize program variables and read */
                             /* initial input from the operator */
  while (cntinue) {          /* do until the operator signals stop */
    indcat = chkinpt ();     /* check for operator input and act on it */
    if (indcat == STOP)      /* if operator signalled stop */
      cntinue = FALSE;       /* set stopping condition */
    else {                   /* otherwise */
/*********************************/
      datagen ();            /* call sensor data simulator */
/*********************************/
      gtdata ();             /* get next data set from sensors */
      bproces ();            /* do basic processing required */
      tempr = 0;             /* set initial level call */
      while (tempr != STOP) /* call level handler */
        tempr = lvlhan (tempr); /* call level handler */
    }
  }
                             /* until all levels processed */
  termnat ();                /* do terminal routines before exiting */
}

  /* Level handler routine */
```

```
lvlhan (valu)
  int valu; {
  int tempr;
  switch (valu) {              /* case statement to direct program control */
    case 0:
      tempr = level0 ();       /* switch control to level 0 */
      break;
    case 1:
      tempr = level1 ();       /* switch control to level 1 */
      break;
    case 2:
      tempr = level2 ();       /* switch control to level 2 */
      break;
    case 3:
      tempr = level3 ();       /* switch control to level 3 */
      break;
    case 4:
      tempr = level4 ();       /* switch control to level 4 */
      break;
    case 5:
      tempr = level5 ();       /* switch control to level 5 */
      break;
    case 6:
      tempr = level6 ();       /* switch control to level 6 */
      break;
    default:
      tempr = STOP;            /* default condition for error */
      break;
  }
  return (tempr);              /* return the next level to be processed */
}

/* This routine initializes applicable list nodes and default arrays */
initlz () {
  int i;
/* initialize sensor ID list */
```

```
  sidhdr = (struct sensid *) calloc (1,(sizeof(struct sensid)));
    /* allocate storage for the header node */
  sidtl = (struct sensid *) calloc (1,(sizeof(struct sensid)));
   /* allocate storage for the sentinel node */
  sidhdr->nxt = sidtl;       /* establish the list */
  sidtl->nxt = NIL;
/* initialize array ID list */
  aidhdr = (struct arayid *) calloc (1,(sizeof(struct arayid)));
    /* allocate storage for the header node */
  aidtl = (struct arayid *) calloc (1,(sizeof(struct arayid)));
   /* allocate storage for the sentinel node */
  aidhdr->next = aidtl;      /* establish the list */
  aidtl->next = NIL;
/* initialize field ID list */
  flidhdr = (struct fildid *) calloc (1,(sizeof(struct fildid)));
    /* allocate storage for the header node */
  flidtl = (struct fildid *) calloc (1,(sizeof(struct fildid)));
   /* allocate storage for the sentinel node */
  flidhdr->follow = flidtl;       /* establish the list */
  flidtl->follow = NIL;
/* initialize deleted sensor ID list */
  dsidhdr = (struct dsensid *) calloc (1,(sizeof(struct dsensid)));
    /* allocate storage for the header node */
  dsidtl = (struct dsensid *) calloc (1,(sizeof(struct dsensid)));
   /* allocate storage for the sentinel node */
  dsidhdr->dnxt = dsidtl;       /* establish the list */
  dsidtl->dnxt = NIL;
/* initialize deleted array ID list */
  daidhdr = (struct darayid *) calloc (1,(sizeof(struct darayid)));
    /* allocate storage for the header node */
  daidtl = (struct darayid *) calloc (1,(sizeof(struct darayid)));
   /* allocate storage for the sentinel node */
  daidhdr->dnext = daidtl;       /* establish the list */
  daidtl->dnext = NIL;
/* initialize deleted field ID list */
  dfidhdr = (struct dfildid *) calloc (1,(sizeof(struct dfildid)));
```

```
    /* allocate storage for the header node */
  dfidtl = (struct dfildid *) calloc (1,(sizeof(struct dfildid)));
   /* allocate storage for the sentinel node */
  dfidhdr->dfollow = dfidtl;      /* establish the list */
  dfidtl->dfollow = NIL;
/* initialize event hypothesis list */
  evhdr = (struct evnt *) calloc (1,(sizeof(struct evnt)));
    /* allocate storage for the header node */
  evtl = (struct evnt *) calloc (1,(sizeof(struct evnt)));
   /* allocate storage for the sentinel node */
  evhdr->fll = evtl;      /* establish the list */
  evtl->last = evhdr;
  evhdr->last = NIL;
  evtl->fll = NIL;
/* initialize classification hypothesis list */
  clhdr = (struct classh *) calloc (1,(sizeof(struct classh)));
    /* allocate storage for the header node */
  cltl = (struct classh *) calloc (1,(sizeof(struct classh)));
   /* allocate storage for the sentinel node */
  clhdr->hnext = cltl;      /* establish the list */
  cltl->hprev = clhdr;
  clhdr->hprev = NIL;
  cltl->hnext = NIL;
/* initialize filtered classification hypothesis list */
  filhdr = (struct filhyp *) calloc (1,(sizeof(struct filhyp)));
    /* allocate storage for the header node */
  filtl = (struct filhyp *) calloc (1,(sizeof(struct filhyp)));
   /* allocate storage for the sentinel node */
  filhdr->fright = filtl;      /* establish the list */
  filtl->fleft = filhdr;
  filhdr->fleft = NIL;
  filtl->fright = NIL;
/*****************************/
/* initialize target track list */
  thead = (struct tgttrk *) calloc (1,(sizeof(struct tgttrk)));
    /* allocate storage for the header node */
```

```
  ttail = (struct tgttrk *) calloc (1,(sizeof(struct tgttrk)));
   /* allocate storage for the sentinel node */
  thead->tnext = ttail;       /* establish the list */
  ttail->tnext = NIL;
/*****************************/
/* initialize root */
  rootptr = (struct fldroot *) calloc (1,(sizeof(struct fldroot)));
    /* allocate storage for root node */
  rootptr->numfld = 0;        /* initialize number of fields */
  for (i=0;i<NBYTES;i++)       /* initialize bit map */
    rootptr->fadajc[i] = 0;
/* initialize field list in root node */
  rootptr->fldhdr = (struct fildnod *) calloc (1,(sizeof(struct fildnod)));
    /* allocate storage for header node */
  rootptr->fldtl = (struct fildnod *) calloc (1,(sizeof(struct fildnod)));
    /* allocate storage for sentinel node */
  (rootptr->fldhdr)->fnext = rootptr->fldtl;      /* establish list */
  (rootptr->fldtl)->fprev = rootptr->fldhdr;
  (rootptr->fldhdr)->fprev = NIL;
  (rootptr->fldtl)->fnext = NIL;
  (rootptr->fldhdr)->fbmapno = NIL;         /* designate as special nodes */
  (rootptr->fldtl)->fbmapno = NPERMAP;
  return;                     /* return to point of call */
}

/* This routine clears all sensor unfiltered data lists, prepares sensor
   window activation records for update, reads the input data from
   the sensors, and creates the unfiltered data lists for sensors if any
   have input */
gtdata () {
  int i, k;
  char thrway;
  struct sensid *p;
  struct sensnod *q, *r;
  clrufdt ();                                 /* clear unfiltered data lists */
  if ((fp = fopen (dfilnam, "r")) == NULL)/* open data file if possible */
```

```
    printf("unable to open data file0);
  else {
    fscanf(fp,"%f",&datatim);                    /* read base time */
    bastim = datatim;                            /* and set it */
    k = (TMWINDW * NPERSEC) / 6;                /* calculate number of bytes */
    k = (((TMWINDW * NPERSEC) % 6) == 0 ? k : (k + 1));
    for (i=0;i<k;i++) {                          /* do for number of bytes */
      p = sidhdr->nxt;                           /* initialize pointer */
      while (p != sidtl) {                       /* do for all sensors */
        switch (((p->indic)->type)/10) {         /* case stmt for sensor type */
          case 1000:
            romnsid (p->indic);                  /* read for minisid s/o */
            break;
          default:
            fscanf(fp,"%c",&thrway);             /* throw away for others */
            break;
        }
        p = p->nxt;                              /* advance pointer */
      }
      bastim =+ (6.0 / NPERSEC);                 /* increment base time */
    }
    fclose (fp);                                 /* close the file */
  }
  return;                                        /* return to point of call */
}

/* This routine clears the unfiltered data lists for all sensors prior
   to reading new unfiltered data, and prepares the sensor window
   historical activation records for update */
clrufdt () {
  int i;
  struct sensid *p;
  struct sensnod *q;
  struct sdatau *r, *temp;
  p = sidhdr->nxt;                               /* initialize pointer */
  while (p != sidtl) {                           /* for all sensors, do */
```

```
    q = p->indic;                                   /* point to sensor node */
    for (i=NWINDW;i>1;i--)                          /* for window list, do */
      q->numcnt[i-1] = q->numcnt[i-2];              /* shift list elements */
    q->numcnt[0] = 0;                               /* initialize new value */
    r = (q->head)->right;                           /* initialize pointer */
    while (r != q->tail) {                          /* while not thru list, do */
      temp = r;                                     /* prepare for removal */
      r = temp->right;                              /* advance pointer */
      dlete1 (temp);                               /* delete from list */
      cfree (temp);                                 /* deallocate storage */
    }
    p = p->nxt;                                     /* advance pointer */
  }
  return;                                           /* return to point of call */
}

/* This routine generates the unfiltered data list for a seismic only
   configuration minisid sensor, and increments the window counter for
   the parent node */
romnsid (q)
  struct sensnod *q; {
  struct sdatau *p;
  char rawdata, temp;
  int j;
  fscanf(fp,"%c",&rawdata);                         /* read a byte of data */
  for (j=1;j<7;j++) {                               /* for each data bit, do */
    temp = rawdata;                                 /* save original data */
    if ((temp =& mask[j]) != 0) {                   /* if an activation */
      p = (struct sdatau *) calloc (1,(sizeof(struct sdatau)));
        /* allocate storage for an unfiltered data node */
      insrt1 (p, q);                               /* insert in the list */
      p->timact = bastim + ((1.0 / NPERSEC) * (j-1)); /* set active time */
      q->numcnt[0] =+ 1;                            /* increment window counter */
    }
  }
  return;                                           /* return to point of call */
```

```
}

/* This routine does the basic numerical processing required by the
   level processors as part of the global data base */
bproces () {
  estexp ();
  chkactv ();
  return;                                   /* return to point of call */
}

/* This routine calculates the smoothed expectation estimate for
   each sensor */
estexp () {
  struct arayid *p;
  struct araynod *q;
  struct sensnod *r;
  int i, val;
  if (systime >= ((float) (TMWINDW * NWINDW))) { /* if able to be done */
    p = aidhdr->next;                       /* point to first list node */
    while (p != aidtl) {                    /* while not at end of list, do */
      q = p->pntr;                          /* point to array */
      r = (q->headr)->nx;                   /* point to first sensor node */
      while (r != q->sntinl) {              /* while not at end of list, do */
        val = 0;                            /* initialize sum */
        for (i=0;i<NWINDW;i++)              /* for each time window, do */
          val =+ r->numcnt[i];              /* calculate sum */
        r->expect =+ (SMOOTH*((float)(val-r->expect))); /* get expect val */
        r = r->nx;                          /* advance node pointer */
      }
      p = p->next;                          /* advance pointer */
    }
  }
  return;                                   /* return to point of call */
}

/* This routine checks to see if an inactive sensor, caused by the user
```

```
   setting the activation time in the future, should be reset to active.
   if so, the sensor is reset */
chkactv () {
  struct sensid *p;
  struct sensnod *q;
  p = sidhdr->nxt;                              /* point to 1st 1st node */
  while (p != sidtl) {                          /* while not through list, do */
    q = p->indic;                               /* point to sensor node */
    if (((systime + q->offon) < ((float)(TMWINDW))) &&
      ((float)fabs(q->offon)) <= systime))  /* if time to activate */
      q->offon = -q->offon;                     /* then do so */
  p = p->nxt;                                   /* advance pointer */
  }
  return;                                       /* return to point of call */
}

/* This routine represents a level of processing in the ground sensor
   AI program */
level0 () {
  int rtnval;
  rtnval = 1;                                     /* set default return value */
  1011r0 ();                                      /* process prod rules */
  1011r1 ();
  1011r2 ();
  1011r3 ();
  return (rtnval);                                /* return next level processed */
}

/* This routine represents a level of processing in the ground sensor
   AI program */
level1 () {
  int rtnval;
  rtnval = 2;                                     /* set default return value */
  1112r1 ();                                    /* process prod rules */
  return (rtnval);                                /* return next level processed */
}
```

```
/* This routine represents a level of processing in the ground sensor
   AI program */
level2 () {
  int rtnval;
  rtnval = 3;                              /* set default return value */
  1213r0 ();                                /* process prod rules */
  1213r1 ();
  1213r2 ();
  1213r3 ();
  return (rtnval);                         /* return next level processed */
}

/* This routine represents a level of processing in the ground sensor
   AI program */
level3 () {
  int rtnval;
  rtnval = 4;                              /* set default return value */
  1313r0 ();                               /* process prod rules */
  1313r1 ();
  1314r0 ();
  return (rtnval);                         /* return next level processed */
}

/* This routine represents a level of processing in the ground sensor
   AI program */
level4 () {
  int rtnval;
  rtnval = 5;                              /* set default return value */
printf("enter level 40);
  1414r0 ();                               /* process prod rules */
  1414r1 ();
  1414r2 ();
  1415r0 ();
  1415r1 ();
  1415r2 ();
```

```
  1415r3 ();
  1415r4 ();
  1415r5 ();
  return (rtnval);                          /* return next level processed */
}

/* This routine represents a level of processing in the ground sensor
   AI program */
level5 () {
  int rtnval;
  rtnval = 6;                               /* set default return value */
  1515r0 ();                                /* process prod rules */
  return (rtnval);                          /* return next level processed */
}

/* This routine represents a level of processing in the ground sensor
   AI program */
level6 () {
  int rtnval;
  rtnval = STOP;                            /* set default return value */
  return (rtnval);                          /* return next level processed */
}

/* This routine performs termnation and cleanup tasks before program
   execution stops */
termnat () {
  return;                                    /* return to point of call */
}
```

```
/* This file contains some primitive routines used frequently by all
   other files in the AI ground sensor program */

#include <stdio.h>
#include "pgmvar.c"          /* include program data */
#include "exincld.c"         /* include external variables */

/* This routine gets a one character response from the user at
   his terminal */
char getans () {
  char ans, temp;
  ans = getchar ();                          /* get user response */
  while ((temp = getchar ()) != '0'); /* ignore all after 1st char */
  printf("0);                               /* skip a line */
  return (ans);                              /* return to point of call */
}

/* This routine gets integer numbers from the user */
getint () {
  char strng[MAXSZ];
  int retval, j;
  j = 0;                                     /* initialize char counter */
  while ((strng[j++] = getchar ()) != '0'); /* get string from user */
  retval = atoi (strng);                     /* convert to integer */
  return (retval);                           /* return integer to point of call */
}

/* This routine gets floating point numbers from the user */
float gtfloat () {
  char strng[MAXSZ];
  float retval;
  int j;
  j = 0;                                     /* initialize char counter */
  while ((strng[j++] = getchar ()) != '0'); /* get string from user */
  retval = (float) atof (strng);             /* convert to floating point */
  return (retval);                           /* return number to point of call */
```

```
}

/* This routine gets the ID number for sensors, arrays, and fields
   that the user wishes to use */
getidnm () {
  int j, nstp, idno;
  nstp = TRUE;                          /* set loop condition to no stop */
  while (nstp) {                        /* while not stop, do */
    j = 0;                              /* initialize char counter */
    printf("enter ID number (0001 - 9999)0);
    idno = getint ();                   /* get ID number */
    if ((idno < 1) || (idno > 9999))    /* if out of limits */
      printf("ID number out of limits0); /* print error message */
    else                                /* otherwise */
      nstp = FALSE;                     /* set stopping condition */
  }
  return (idno);                        /* return ID to point of call */
}
```

```
/* This collection of routines provides the basic means of node
   insertion for the nodes of the AI ground sensor program.  Routines
   1, 2, 9, 10, 12 to 14, 16, 17, and 22 are insertion routines
   for doubly linked lists.  These routines expect that the node
   to be inserted is identified as the first parameter passed,
   while the second parameter passed is a pointer to the ''parent''
   of the list, which in turn contains a pointer to the sentinel (last)
   node of the list.  As an additional feature, these routines link the
   new node to the parent of the list.  Routines 3, 5, 7, 11, 15, and 18 are
   also insertion routines for doubly linked lists.  The first parameter
   passed is a pointer to the node to be inserted, while the second
   parameter is a pointer to the node before which the insertion will
   take place.  In both of the above cases, the insertion attempt is
   assumed to be valid; hence nothing is returned from the routines.
   Routines 4, 6, 8, and 19 to 21 are inertion routines for singly linked
   lists.  The first parameter passed identifies the node to be inserted,
   the second parameter is a pointer to the node at the head of the list,
   and the third parameter is a pointer to the last node in the list.
   The list is searched node by node until the proper insertion point is
   found (in which case the node is inserted and a ''valid'' indicator is
   passed back to the point of call) or until an attempt to insert a
   duplicate node is discovered (in which case the node is not inserted
   and an ''invalid'' indicator is passed back to the point of call). */

#include <stdio.h>
#include "pgmvar.c"          /* include program data */
#include "exincld.c"         /* include external variables */

  /* Insert routine for an unfiltered sensor data node */
insrt1 (p,q)
  struct sdatau *p;
  struct sensnod *q; {
  p->right = q->tail;              /* set right pointer of new node */
  p->left = (q->tail)->left;       /* set left pointer of new node */
  (q->tail)->left = p;             /* reset next node left pointer */
  (p->left)->right = p;            /* reset previous node right pointer */
```

```
  p->prnt = q;                    /* set new node parent pointer */
  return;
}

  /* Insert routine for a filtered sensor data node */
insrt2 (p,q)
  struct sdataf *p;
  struct sdatafl *q; {
  p->rght = q->dfsent;            /* set right pointer of new node */
  p->lft = (q->dfsent)->lft;      /* set left pointer of new node */
  (q->dfsent)->lft = p;           /* reset next node left pointer */
  (p->lft)->rght = p;             /* reset previous node right pointer */
  p->pnt = q;                     /* set new node parent pointer */
  return;
}

  /* Insert routine for a sensor node */
insrt3 (p,q)
  struct sensnod *p, *q; {
  p->nx = q;                      /* set next pointer of new node */
  p->prev = q->prev;              /* set previous pointer of new node */
  q->prev = p;                    /* reset next node previous pointer */
  (p->prev)->nx = p;              /* reset previous node next pointer */
  return;
}

  /* Insert routine for a sensor id list node */
insrt4 (p,q,r)
  struct sensid *p, *q, *r; {
  struct sensid *s;
  r->idno = p->idno;              /* establish EOL stopping condition */
  s = q->nxt;                     /* initialize pointer */
  while (s->idno < p->idno) {     /* advance in list while */
    q = s;                        /* insert position not found */
    s = q->nxt; }
  if ((s->idno == p->idno) && (s != r))/* id already used ? */
```

```
    return (ERROR);                 /* yes, return ERROR */
  else {
    p->nxt = s;                     /* no, insert node */
    q->nxt = p;
  }
  return (VALID);                 /* and return successful completion */
}

  /* Insert routine for a sensor array node */
insrt5 (p,q)
  struct araynod *p, *q; {
  p->rt = q;                        /* set right pointer of new node */
  p->lf = q->lf;                    /* set left pointer of new node */
  q->lf = p;                        /* reset next node left pointer */
  (p->lf)->rt = p;                  /* reset previous node right pointer */
  return;
}

  /* Insert routine for a sensor array id list node */
insrt6 (p,q,r)
  struct arayid *p, *q, *r; {
  struct arayid *s;
  r->araynum = p->araynum;          /* establish EOL stopping condition */
  s = q->next;                      /* initialize pointer */
  while (s->araynum < p->araynum) {  /* advance in list while */
    q = s;                          /* insert position not found */
    s = q->next; }
  if ((s->araynum == p->araynum) && (s != r))/* id already used ? */
    return (ERROR);                 /* yes, return ERROR */
  else {
    p->next = s;                     /* no, insert node */
    q->next = p;
  }
  return (VALID);                 /* and return successful completion */
}
```

```
  /* Insert routine for a sensor field node */
insrt7 (p,q)
  struct fildnod *p, *q; {
  p->fnext = q;                    /* set new node next pointer */
  p->fprev = q->fprev;             /* set new node previous pointer */
  q->fprev = p;                    /* reset next node previous pointer */
  (p->fprev)->fnext = p;           /* reset previous node next pointer */
  return;
}

  /* Insert routine for a sensor field id list node */
insrt8 (p,q,r)
  struct fildid *p, *q, *r; {
  struct fildid *s;
  r->fildnum = p->fildnum;         /* establish EOL stopping condition */
  s = q->follow;                   /* initialize pointer */
  while (s->fildnum < p->fildnum) {  /* advance in list while */
    q = s;                         /* insert position not found */
    s = q->follow; }
  if ((s->fildnum == p->fildnum) && (s != r))/* id already used ? */
    return (ERROR);                /* yes, return ERROR */
  else {
    p->follow = s;                 /* no, insert node */
    q->follow = p;
  }
  return (VALID);                /* and return successful completion */
}

  /* Insert routine for a valid set node */
insrt9 (p,q)
  struct vset *p;
  struct sensnod *q; {
  p->foll = q->sentl;              /* set right pointer of new node */
  p->prevs = (q->sentl)->prevs;/* set left pointer of new node */
  (q->sentl)->prevs = p;           /* reset next node left pointer */
  (p->prevs)->foll = p;            /* reset previous node right pointer */
```

```
  p->lnk = q;                    /* set new node parent pointer */
  return;
}

  /* Insert routine for an event hypothesis list node */
insrt10 (p,q)
  struct evntl *p;
  struct vset *q; {
  p->follw = q->eltail;          /* set right pointer of new node */
  p->prvious = (q->eltail)->prvious;   /* set left pointer of new node */
  (q->eltail)->prvious = p;      /* reset next node left pointer */
  (p->prvious)->follw = p;       /* reset previous node right pointer */
  p->vlower = q;                 /* set new node parent pointer */
  return;
}

  /* Insert routine for an event hypothesis node */
insrt11 (p,q)
  struct evnt *p, *q; {
  p->fll = q;                    /* set new node next pointer */
  p->last = q->last;             /* set new node previous pointer */
  q->last = p;                   /* reset next node previous pointer */
  (p->last)->fll = p;            /* reset previous node next pointer */
  return;
}

  /* Insert routine for a velocity list node */
insrt12 (p,q)
  struct velist *p;
  struct evnt *q; {
  p->vnext = q->vsent;           /* set right pointer of new node */
  p->vlast = (q->vsent)->vlast;/* set left pointer of new node */
  (q->vsent)->vlast = p;         /* reset next node left pointer */
  (p->vlast)->vnext = p;         /* reset previous node right pointer */
  p->vpar = q;                   /* set new node parent pointer */
  return;
```

```
}

  /* Insert routine for a valid set list node */
insrt13 (p,q)
  struct vsetl *p;
  struct evnt *q; {
  p->lfoll = q->vstail;          /* set right pointer of new node */
  p->lprev = (q->vstail)->lprev;   /* set left pointer of new node */
  (q->vstail)->lprev = p;        /* reset next node left pointer */
  (p->lprev)->lfoll = p;         /* reset previous node right pointer */
  p->eparnt = q;                 /* set new node parent pointer */
  return;
}

  /* Insert routine for a classification hypothesis list node */
insrt14 (p,q)
  struct chypl *p;
  struct evnt *q; {
  p->cnxt = q->chtail;           /* set right pointer of new node */
  p->cprev = (q->chtail)->cprev;   /* set left pointer of new node */
  (q->chtail)->cprev = p;        /* reset next node left pointer */
  (p->cprev)->cnxt = p;          /* reset previous node right pointer */
  p->cparnt = q;                 /* set new node parent pointer */
  return;
}

  /* Insert routine for a classification hypothesis node */
insrt15 (p,q)
  struct classh *p, *q; {
  p->hnext = q;                  /* set new node next pointer */
  p->hprev = q->hprev;           /* set new node previous pointer */
  q->hprev = p;                  /* reset next node previous pointer */
  (p->hprev)->hnext = p;         /* reset previous node next pointer */
  return;
}
```

```
  /* Insert routine for a class hypothesis event list node */
insrt16 (p,q)
  struct eventl *p;
  struct classh *q; {
  p->elnext = q->chtl;          /* set right pointer of new node */
  p->elprev = (q->chtl)->elprev;   /* set left pointer of new node */
  (q->chtl)->elprev = p;        /* reset next node left pointer */
  (p->elprev)->elnext = p;      /* reset previous node right pointer */
  p->chparnt = q;               /* set new node parent pointer */
  return;
}

  /* Insert routine for a filtered class hypothesis list node */
insrt17 (p,q)
  struct fhypl *p;
  struct classh *q; {
  p->flnext = q->fhtail;        /* set right pointer of new node */
  p->flprev = (q->fhtail)->flprev; /* set left pointer of new node */
  (q->fhtail)->flprev = p;      /* reset next node left pointer */
  (p->flprev)->flnext = p;      /* reset previous node right pointer */
  p->clprnt = q;                /* set new node parent pointer */
  return;
}

  /* Insert routine for a filtered class hypothesis node */
insrt18 (p,q)
  struct filhyp *p, *q; {
  p->fright = q;                /* set new node next pointer */
  p->fleft = q->fleft;          /* set new node previous pointer */
  q->fleft = p;                 /* reset next node previous pointer */
  (p->fleft)->fright = p;       /* reset previous node next pointer */
  return;
}

  /* Insert routine for a deleted sensor id list node */
insrt19 (p,q,r)
```

```
  struct dsensid *p, *q, *r; {
  struct dsensid *s;
  r->didno = p->didno;          /* establish EOL stopping condition */
  s = q->dnxt;                  /* initialize pointer */
  while (s->didno < p->didno) {/* advance in list while */
    q = s;                      /* insert position not found */
    s = q->dnxt; }
  if ((s->didno == p->didno) && (s != r))/* id already used ? */
    return (ERROR);             /* yes, return ERROR */
  else {
    p->dnxt = s;                /* no, insert node */
    q->dnxt = p;
  }
  return (VALID);             /* and return successful completion */
}

  /* Insert routine for a deleted sensor array id list node */
insrt20 (p,q,r)
  struct darayid *p, *q, *r; {
  struct darayid *s;
  r->daraynum = p->daraynum;    /* establish EOL stopping condition */
  s = q->dnext;                 /* initialize pointer */
  while (s->daraynum < p->daraynum) {  /* advance in list while */
    q = s;                      /* insert position not found */
    s = q->dnext; }
  if ((s->daraynum == p->daraynum) && (s != r))/* id already used ? */
    return (ERROR);             /* yes, return ERROR */
  else {
   p->dnext = s;                /* no, insert node */
   q->dnext = p;
  }
  return (VALID);             /* and return successful completion */
}

  /* Insert routine for a deleted sensor field id list node */
insrt21 (p,q,r)
```

```
  struct dfildid *p, *q, *r; {
  struct dfildid *s;
  r->dfildnum = p->dfildnum;    /* establish EOL stopping condition */
  s = q->dfollow;               /* initialize pointer */
  while (s->dfildnum < p->dfildnum) {  /* advance in list while */
    q = s;                      /* insert position not found */
    s = q->dfollow; }
  if ((s->dfildnum == p->dfildnum) && (s != r))/* id already used ? */
    return (ERROR);             /* yes, return ERROR */
  else {
    p->dfollow = s;             /* no, insert node */
    q->dfollow = p;
  }
  return (VALID);               /* and return successful completion */
}

  /* Insert routine for a filtered sensor data list list node */
insrt22 (p,q)
  struct sdataf1 *p;
  struct sensnod *q; {
  p->fdlnx = q->t1;             /* set right pointer of new node */
  p->fdlpr = (q->t1)->fdlpr;    /* set left pointer of new node */
  (q->t1)->fdlpr = p;           /* reset next node left pointer */
  (p->fdlpr)->fdlnx = p;        /* reset previous node right pointer */
  p->dlprnt = q;                /* set new node parent pointer */
  return;
}
```

```
/* This collection of routines provides the basic means of node
   deletion for the nodes of the AI ground sensor program.
   Routines 1 to 3, 5, 7, 9 to 18, and 22 are deletion routines for doubly
   linked lists.  The parameter passed is a pointer to the node to be
   deleted.  Routines 4, 6, 8, and 19 to 21 are deletion routines for singly
   linked lists.  The first parameter passed is a pointer to a node
   immediately before the node to be deleted.  The second parameter
   passed is a pointer to the node to be deleted. */

#include <stdio.h>
#include "pgmvar.c"          /* include program data */
#include "exincld.c"         /* include external variables */

  /* Delete routine for an unfiltered sensor data node */
dlete1 (p)
  struct sdatau *p; {
  (p->left)->right = p->right;      /* reset previous node pointer */
  (p->right)->left = p->left;       /* reset next node pointer */
  return;
}

  /* Delete routine for a filtered sensor data node */
dlete2 (p)
  struct sdataf *p; {
  (p->lft)->rght = p->rght;         /* reset previous node pointer */
  (p->rght)->lft = p->lft;          /* reset next node pointer */
  return;
}

  /* Delete routine for a sensor node */
dlete3 (p)
  struct sensnod *p; {
  (p->prev)->nx = p->nx;            /* reset previous node pointer */
  (p->nx)->prev = p->prev;          /* reset next node pointer */
  return;
}
```

```
  /* Delete routine for a sensor id list node */
dlete4 (q,p)
  struct sensid *q, *p; {
  q->nxt = p->nxt;                    /* reset previous node pointer */
  return;
}

  /* Delete routine for a sensor array node */
dlete5 (p)
  struct araynod *p; {
  (p->lf)->rt = p->rt;                /* reset previous node pointer */
  (p->rt)->lf = p->lf;                /* reset next node pointer */
  return;
}

  /* Delete routine for a sensor array id list node */
dlete6 (q,p)
  struct arayid *q, *p; {
  q->next = p->next;                  /* reset previous node pointer */
  return;
}

  /* Delete routine for a sensor field node */
dlete7 (p)
  struct fildnod *p; {
  (p->fprev)->fnext = p->fnext;       /* reset previous node pointer */
  (p->fnext)->fprev = p->fprev;       /* reset next node pointer */
  return;
}

  /* Delete routine for a sensor field id list node */
dlete8 (q,p)
  struct fildid *q, *p; {
  q->follow = p->follow;              /* reset previous node pointer */
  return;
```

```
}

  /* Delete routine for a valid set node */
dlete9 (p)
  struct vset *p; {
  (p->prevs)->foll = p->foll;       /* reset previous node pointer */
  (p->foll)->prevs = p->prevs;      /* reset next node pointer */
  return;
}

  /* Delete routine for an event hypothesis list node */
dlete10 (p)
  struct evntl *p; {
  (p->prvious)->follw = p->follw;  /* reset previous node pointer */
  (p->follw)->prvious = p->prvious;/* reset next node pointer */
  return;
}

  /* Delete routine for an event hypothesis node */
dlete11 (p)
  struct evnt *p; {
  (p->last)->fll = p->fll;          /* reset previous node pointer */
  (p->fll)->last = p->last;         /* reset next node pointer */
  return;
}

  /* Delete routine for a velocity list node */
dlete12 (p)
  struct velist *p; {
  (p->vlast)->vnext = p->vnext;     /* reset previous node pointer */
  (p->vnext)->vlast = p->vlast;     /* reset next node pointer */
  return;
}

  /* Delete routine for a valid set list node */
dlete13 (p)
```

```
  struct vsetl *p; {
  (p->lprev)->lfoll = p->lfoll;     /* reset previous node pointer */
  (p->lfoll)->lprev = p->lprev;     /* reset next node pointer */
  return;
}

  /* Delete routine for a classification hypothesis list node */
dlete14 (p)
  struct chypl *p; {
  (p->cprev)->cnxt = p->cnxt;       /* reset previous node pointer */
  (p->cnxt)->cprev = p->cprev;      /* reset next node pointer */
  return;
}

  /* Delete routine for a classification hypothesis node */
dlete15 (p)
  struct classh *p; {
  (p->hprev)->hnext = p->hnext;     /* reset previous node pointer */
  (p->hnext)->hprev = p->hprev;     /* reset next node pointer */
  return;
}

  /* Delete routine for a class hypothesis event list node */
dlete16 (p)
  struct eventl *p; {
  (p->elprev)->elnext = p->elnext; /* reset previous node pointer */
  (p->elnext)->elprev = p->elprev; /* reset next node pointer */
  return;
}

  /* Delete routine for a filtered class hypothesis list node */
dlete17 (p)
  struct fhypl *p; {
  (p->flprev)->flnext = p->flnext; /* reset previous node pointer */
  (p->flnext)->flprev = p->flprev; /* reset next node pointer */
  return;
```

```
}

  /* Delete routine for a filtered class hypothesis node */
dlete18 (p)
  struct filhyp *p; {
  (p->fleft)->fright = p->fright;  /* reset previous node pointer */
  (p->fright)->fleft = p->fleft;   /* reset next node pointer */
  return;
}

  /* Delete routine for a deleted sensor id list node */
dlete19 (q,p)
  struct dsensid *q, *p; {
  q->dnxt = p->dnxt;               /* reset previous node pointer */
  return;
}

  /* Delete routine for a deleted sensor array id list node */
dlete20 (q,p)
  struct darayid *q, *p; {
  q->dnext = p->dnext;             /* reset previous node pointer */
  return;
}

  /* Delete routine for a deleted sensor field id list node */
dlete21 (q,p)
  struct dfildid *q, *p; {
  q->dfollow = p->dfollow;         /* reset previous node pointer */
  return;
}

  /* Delete routine for a filtered sensor data list list node */
dlete22 (p)
  struct sdatafl *p; {
  (p->fdlpr)->fdlnx = p->fdlnx;        /* reset previous node pointer */
  (p->fdlnx)->fdlpr = p->fdlpr;         /* reset next node pointer */
```

```
    return;
}
```

```
/* This module contains operations necessary to complete the
   insertion of a sensor and operations performed on sensor nodes */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "genvar.c"                          /* include generator data */
#include "exincld.c"                         /* include external variables */

/* This routine inserts a new sensor into the parent array sensor list
   in its proper position, links the new sensor to the parent array,
   increments the number of sensors counter in the array, or returns
   an error code if the insert was not possible */
instsnd (new, arrayIp)
  struct sensnod *new;
  struct arayid *arrayIp; {
  struct sensnod *r, *s;
  int rtnval;
  new->parent = arrayIp->pntr;               /* link to parent node */
  if (((arrayIp->pntr)->numsen) == NPERMAP) /* if parent already full */
    rtnval = ERROR;                          /* set error condition */
  else {                                     /* otherwise */
    rtnval = VALID;                          /* insertion is possible */
    r = (arrayIp->pntr)->headr;              /* initialize search pointers */
    s = r->nx;
    while ((r->sbmapno) == ((s->sbmapno)-1)) { /* if insrt pos not fnd */
      r = s;                                 /* advance pointers */
      s = r->nx;
    }
    insrt3 (new, s);                        /* insert node in the list */
    new->sbmapno = r->sbmapno + 1;           /* set bit map index */
    (arrayIp->pntr)->numsen =+ 1;            /* increment no of arrays cntr */
  }
  return (rtnval);                           /* return status indicator */
}

/* This routine creates a new unfiltered data list given a pointer to the
```

```
   parent sensor node */
intusdl (new)
  struct sensnod *new; {
  struct sdatau *first, *last;
  first = (struct sdatau *) calloc (1,(sizeof(struct sdatau)));
    /* allocate storage for the header node */
  last = (struct sdatau *) calloc (1,(sizeof(struct sdatau)));
    /* allocate storage for the sentinel node */
  new->head = first;                          /* set parent links */
  new->tail = last;
  first->left = NIL;                          /* establish list */
  first->right = last;
  last->left = first;
  last->right = NIL;
  first->prnt = new;                          /* link to parent */
  last->prnt = new;
  /* designate as special nodes */
  return;                                     /* return to point of call */
}

/* This routine creates a new filtered data list list given a pointer to
   the parent sensor node */
infsdll (new)
  struct sensnod *new; {
  struct sdatafl *first, *last;
  first = (struct sdatafl *) calloc (1,(sizeof(struct sdatafl)));
    /* allocate storage for the header node */
  last = (struct sdatafl *) calloc (1,(sizeof(struct sdatafl)));
    /* allocate storage for the sentinel node */
  new->hd = first;                            /* set parent links */
  new->tl = last;
  first->fdlpr = NIL;                           /* establish list */
  first->fdlnx = last;
  last->fdlpr = first;
  last->fdlnx = NIL;
  first->dlprnt = new;                           /* link to parent */
```

```
  last->dlprnt = new;
  /* designate as special nodes */
  return;                                   /* return to point of call */
}

/* This routine creates a new valid set list given a pointer to the
   parent sensor node */
initvsl (new)
  struct sensnod *new; {
  struct vset *first, *last;
  first = (struct vset *) calloc (1,(sizeof(struct vset)));
    /* allocate storage for the header node */
  last = (struct vset *) calloc (1,(sizeof(struct vset)));
    /* allocate storage for the sentinel node */
  new->hdr = first;                         /* set parent links */
  new->sentl = last;
  first->prevs = NIL;                       /* establish list */
  first->foll = last;
  last->prevs = first;
  last->foll = NIL;
  first->lnk = new;                         /* link to parent */
  last->lnk = new;
  /* designate as special nodes */
  return;                                   /* return to point of call */
}

/* This routine checks to see if a newly created sensor is adjacent to
   any other already existent sensors, given a pointer to the new sensor
   and the parent array node */
sadjcnt (new, p)
  struct sensnod *new;
  struct arayid *p; {
  int nstp, n, j, idno;
  char ans;
  struct sensid *ptr, *chkslst ();
  printf("is this sensor adjacent to any already0); /* print user */
```

```
  printf("created sensors ? (y or n)");            /* instructions */
  ans = getans ();                                 /* get user response */
  if (ans == 'y') {                                /* if yes */
    nstp = TRUE;                                   /* set loop condition to no stop */
    while (nstp) {                                 /* while not stop, do */
      printf("enter number of sensors the0); /* print user instructions */
      printf("new sensor is adjacent to");
      n = getint ();                               /* get user response */
      if ((n < 1) || (n > NPERMAP - 1)) /* if out of bounds */
        printf("number out of bounds0);/* print error message */
      else                                         /* otherwise */
        nstp = FALSE;                              /* good value */
    }
    for (j=0;j<n;j++) {                            /* for the number given, do */
      nstp = TRUE;                                 /* set loop condition to no stop */
      while (nstp) {                               /* while not stop, do */
        printf("for sensor %d adjacent to the new sensor0,j+1);
          /* print user instructions */
        idno = getidnm ();                         /* get ID number */
        if((ptr = chkslst (idno)) == NIL) { /* if no associated sensor */
          printf("no sensor associated with0); /* print error msg */
          printf("ID number given0);
        }
        else                                       /* otherwise */
          nstp = FALSE;                            /* good input */
      }
      stsbmap (new->sbmapno,((ptr->indic)->sbmapno),p); /* set bit map */
    }
  }
  return;                                          /* return to point of call */
}

/* This routine checks to see if a sensor exists given a sensor number
   from the user.  If it exists it returns a pointer to the ID list node */
struct sensid *chkslst (id)
  int id; {
```

```
  struct sensid *p;
  sidtl->idno = id;                          /* set stopping condition */
  p = sidhdr->nxt;                             /* initialize search pointer */
  while ((p->idno) != id)                    /* while not found, do */
    p = p->nxt;                                /* advance pointer */
  if (p == sidtl)                              /* if not found */
    return (NIL);                              /* return error condition */
  else                                         /* otherwise */
    return (p);                                /* return pointer to ID node */
}

/* This routine sets the sensor list bit map given the two indices and
   a pointer to the ID of the parent node */
stsbmap (i, j, p)
  int i, j;
  struct arayid *p; {
  int bytindx, mskindx, temp, k;
  for (k=0;k<2;k++) {                          /* do twice */
    bytindx = NPRUNIT * i + j/8;               /* calculate byte index */
    mskindx = j % 8;                           /* calculate offset */
    (p->pntr)->sadajc[bytindx] =| mask[mskindx]; /* set bit map */
    temp = i;                                  /* now reverse indices */
    i = j;
    j = temp;                                  /* and repeat the process */
  }
  return;                                      /* return to point of call */
}

/* This routine gets sensor characteristics from the user, inserts the
   sensor ID number into the sensor ID list, and links the ID list node
   with the sensor node */
gtsnval (new, sensnbr)
  struct sensnod *new;
  int sensnbr; {
  char ans;
  int val, nstp;
```

```
  struct sensid *p;
  nstp = TRUE;                              /* set loop condition to no stop */
  p = (struct sensid *) calloc (1,(sizeof(struct sensid)));
    /* allocate storage for new ID node */
  p->idno = sensnbr;                       /* initialize ID */
  while ((val = insrt4 (p,sidhdr,sidtl)) != VALID) { /* while no insert */
    printf("the sensor ID number already used0); /* print error msg */
    printf("enter another0);                /* request another ID */
    p->idno = getidnm ();                   /* get user response */
  }
  p->indic = new;                           /* link ID to node */
  new->sidback = p;                         /* link node to ID */
/* get sensor type and initialization values peculiar to the type */
  while (nstp) {                            /* while not stop, do */
    printf("enter sensor type");            /* ask for type */
    val = getint ();                        /* get user response */
    switch (val) {                          /* case stmt for answer */
      case 1000:
        somnsid (new, val);                 /* initialization for minisid s/o */
        nstp = FALSE;                       /* set stopping condition */
        break;
      default:
        printf("type not recognized0);/* print error message */
/*****************************************/
        tempfix (new, val);                 /* accept for debug only */
        nstp = FALSE;                       /* set stopping condition */
/*****************************************/
        break;
    }
  }
/* now for user-defined initialization common to all sensors */
  printf("is the new sensor a boundary sensor ? (y or n)"); /* user instr */
  ans = getans ();                          /* get user response */
  if (ans == 'y') {                         /* if yes */
    new->type =+ 1;                         /* set boundary indicator */
    printf("as a reminder, if the addition of this boundary sensor0);
```

```
  printf("causes other sensor characteristics to change you must0);
  printf("change them...0);          /* print user reminder */
}
nstp = TRUE;                         /* set loop condition to no stop */
while (nstp) {                       /* while not stop, do */
  printf("enter sensor UTM X coord (4 digit)"); /* ask for location */
  new->slocx = gtfloat ();           /* get X coord */
  if ((new->slocx < MINMAPX) || (new->slocx > MAXMAPX)) /* in limits? */
    printf("X coord out of limits0); /* no, print error message */
  else                               /* otherwise */
    nstp = FALSE;                    /* set stopping condition */
}
nstp = TRUE;                         /* set loop condition to no stop */
while (nstp) {                       /* while not stop, do */
  printf("enter sensor UTM Y coord (4 digit)"); /* ask for location */
  new->slocy = gtfloat ();           /* get Y coord */
  if ((new->slocy < MINMAPY) || (new->slocy > MAXMAPY)) /* in limits? */
    printf("Y coord out of limits0); /* no, print error message */
  else                               /* otherwise */
    nstp = FALSE;                    /* set stopping condition */
}
if (new->accangl < 360.0) {          /* if a directional sensor */
  nstp = TRUE;                       /* set loop condition to no stop */
  while (nstp) {                     /* while not stop, do */
    printf("enter detection azimuth (deg)"); /* user instruction */
    new->detdir = gtfloat ();        /* get azimuth */
    if ((new->detdir < 0.0) || (new->detdir > 360.0)) /* in limits ? */
      printf("azimuth not in limits0); /* no, print error message */
    else                             /* otherwise */
      nstp = FALSE;                  /* set stopping condition */
  }
}
printf("default detection radii by type have been loaded0); /* user */
printf("for the sensor.  Changes required ? (y or n)"); /* instruction */
ans = getans ();                     /* get user response */
if (ans == 'y')                      /* if yes */
```

```
    drchg (new);                              /* change detection radii */
  if (new->accangl != 360.0) {                /* if a directional sensor */
    printf("a default acceptance angle has been loaded0); /* user */
    printf("for the sensor.  Changes required ? (y or n)"); /* instr */
    ans = getans ();                          /* get user response */
    if (ans == 'y')                           /* if yes */
      aangchg (new);                         /* change acceptance angle */
  }
  printf("the default time of activation is the present0); /* user */
  printf("system time.  Change required ? (y or n)"); /* instr */
  ans = getans ();                            /* get user response */
  if (ans == 'y')                             /* if yes */
    tmacchg (new);                           /* change activation time */
  if (new->tmactiv > systime)                 /* if user has deactivated sensr */
    new->offon = -new->tmactiv;               /* set deactivation indicator */
  printf("default inhibit time has been loaded for0); /* user instr */
  printf("the sensor for its type.  Change required ? (y or n)");
  ans = getans ();                            /* get user response */
  if (ans == 'y')                             /* if yes */
    inhbchg (new);                           /* change inhibit time */
  printf("default soil conditions have been loaded0); /* user instr */
  printf("for the sensor.  Change required ? (y or n)");
  ans = getans ();                            /* get user response */
  if (ans == 'y')                             /* if yes */
    soilchg (new);                            /* change soil code */
  printf("default programmed EOL has been loaded for0); /* user instr */
  printf("the sensor by its type.  Change required ? (y or n)");
  ans = getans ();                            /* get user response */
  if (ans == 'y')                             /* if yes */
    eolchg (new);                             /* change EOL */
  return (p->idno);                           /* return valid array ID */
}

/* This routine loads the default values peculiar to the minisid in
   the "seismic only" mode */
somnsid (new,val)
```

```
  struct sensnod *new;
  int val; {
  new->type = val * 10;                          /* set type */
  new->accangl = 360.0;                          /* set for omni-directional */
  new->hs = 10.0;                                /* set inhibit time */
  new->coloc = NIL;                              /* no colocated sensor */
  new->detrad[0] = 30.0;                         /* detection radius table */
  new->detrad[1] = 100.0;
  new->detrad[2] = 500.0;
  new->detrad[3] = 550.0;
  new->detrad[4] = 700.0;
  new->detrad[5] = 950.0;
  new->eol = new->tmactiv + 4320000.0;       /* set EOL */
  return;                                        /* return to point of call */
}

/* This routine allows the user to make changes in the detection radius
   table for a sensor */
drchg (new)
  struct sensnod *new; {
  int i, nstp;
  nstp = TRUE;                                /* set loop condition to no stop */
  printf("changes to detection radii are made by type0); /* user instr */
  printf("and may be made until an invalid type number is input0);
  while (nstp) {                              /* while not stop, do */
    printf("enter target type");              /* user instructions */
    i = getint ();                            /* get user response */
    if ((i < 1) || (i > NTTYP))               /* if type out of limits */
      nstp = FALSE;                           /* set stopping condition */
    else {                                    /* otherwise */
      printf("detection radius for type %d0,i); /* user instr */
      printf("is currently %f0,new->detrad[i-1]);
      printf("enter new value (meters)"); /* get new value */
      new->detrad[i-1] = gtfloat ();          /* store */
      printf("new value is %f0,new->detrad[i-1]); /* echo to user */
    }
```

```
  }
  return;                                   /* return to point of call */
}

/* This routine allows the user to change the acceptance angle for
   a sensor, given a pointer to the sensor node */
aangchg (new)
  struct sensnod *new; {
  int nstp;
  nstp = TRUE;                              /* set loop condition to no stop */
  printf("current value for acceptance angle is %f0,new->accangl);
    /* print user instructions */
  while (nstp) {                            /* while not stop, do */
    printf("enter new value (degrees)");    /* ask for change */
    new->accangl = gtfloat ();              /* get user response */
    if ((new->accangl < 0.0) || (new->accangl > 360.0)) /* in limits ? */
      printf("acceptance angle out of limits0); /* no, print error msg */
    else {                                  /* otherwise */
      nstp = FALSE;                         /* set stopping condition */
      printf("new value is %f0,new->accangl); /* echo new value */
    }
  }
  return;                                   /* return to point of call */
}

/* This routine allows the user to change the time of activation for
   a sensor, given a pointer to the sensor node */
tmacchg (new)
  struct sensnod *new; {
  int nstp;
  nstp = TRUE;                              /* set loop condition to no stop */
  printf("current value for activation time is %f0,new->tmactiv);
    /* print user instructions */
  while (nstp) {                            /* while not stop, do */
    printf("enter new value (seconds)");    /* ask for change */
    new->tmactiv = gtfloat ();              /* get user response */
```

```
    if (new->tmactiv < 0.0)                /* in limits ? */
      printf("activation time out of limits0); /* no, print error msg */
    else {                                 /* otherwise */
      nstp = FALSE;                        /* set stopping condition */
      printf("new value is %f0,new->tmactiv); /* echo new value */
    }
  }
  return;                                  /* return to point of call */
}

/* This routine allows the user to change the inhibit time for
   a sensor, given a pointer to the sensor node */
inhbchg (new)
  struct sensnod *new; {
  int nstp;
  nstp = TRUE;                             /* set loop condition to no stop */
  printf("current value for inhibit time is %f0,new->hs);
    /* print user instructions */
  while (nstp) {                           /* while not stop, do */
    printf("enter new value (seconds)"); /* ask for change */
    new->hs = gtfloat ();                 /* get user response */
    if (new->hs < 0.0)                     /* in limits ? */
      printf("inhibit time out of limits0); /* no, print error msg */
    else {                                 /* otherwise */
      nstp = FALSE;                        /* set stopping condition */
      printf("new value is %f0,new->hs); /* echo new value */
    }
  }
  return;                                  /* return to point of call */
}

/* This routine allows the user to change the soil code for
   a sensor, given a pointer to the sensor node */
soilchg (new)
  struct sensnod *new; {
  int nstp;
```

```
  nstp = TRUE;                                /* set loop condition to no stop */
  printf("current value for soil code is %d0,new->soil);
    /* print user instructions */
  while (nstp) {                              /* while not stop, do */
    printf("enter new value ");               /* ask for change */
    new->soil = getint ();                    /* get user response */
    if (new->soil < 0)                        /* in limits ? */
      printf("soil code out of limits0); /* no, print error msg */
    else {                                    /* otherwise */
      nstp = FALSE;                           /* set stopping condition */
      printf("new value is %d0,new->soil); /* echo new value */
    }
  }
  return;                                     /* return to point of call */
}

/* This routine allows the user to change the end of life value for
   a sensor, given a pointer to the sensor node */
eolchg (new)
  struct sensnod *new; {
  int nstp;
  nstp = TRUE;                                /* set loop condition to no stop */
  printf("current value for EOL offset is %f0,(new->eol-new->tmactiv));
    /* print user instructions */
  while (nstp) {                              /* while not stop, do */
    printf("enter new offset value (seconds)"); /* ask for change */
    new->eol = gtfloat ();                   /* get user response */
    if (new->eol < 0,0)                       /* in limits ? */
      printf("EOL out of limits0);          /* no, print error msg */
    else {                                    /* otherwise */
      nstp = FALSE;                           /* set stopping condition */
      printf("new value is %f0,new->eol); /* echo new value */
      new->eol = new->tmactiv + new->eol; /* reset eol */
      printf("programmed EOL at %f0,new->eol);
    }
  }
```

```
  return;                                   /* return to point of call */
}

/* This routine loads the adjacent sensor bit map numbers and the
   adjacency count into an array, given a pointer to the array node, the
   target sensor index number, and a pointer to the array into which
   the results will be loaded */
gtadsen (p, q, r)
  int p, r[];
  struct araynod *q; {
  char temp;
  int i, j, k, l;
  for (i=0;i<NPERMAP;i++)                   /* zero results array */
    r[i] = 0;
  i = NPRUNIT * p;                          /* calculate base byte index */
  l = 1;                                    /* initialize array index */
  for (j=0;j<NPRUNIT;j++)                   /* for all bytes */
    for (k=0;k<8;k++) {                     /* for all bits */
      temp = q->sadajc [i+j];               /* target byte to working stroage */
      temp =& (~(~mask[k]));                /* isolate target bit */
      if (temp != 0) {                      /* if bit was on */
        r[l] = j * 8 + k;                   /* load sensor index into array */
        l =+ 1;                             /* increment array index */
      }
    }
  r[0] = l - 1;                             /* load count into array */
  return;
}

/* This routine gets the first boundary sensor it encounters in an array,
   given a pointer to the parent array.  It returns the bit map number
   of the located boundary sensor */
gtbsnsr (q)
  struct araynod *q; {
  int i, j, k, l, count;
  char temp;
```

```
    i = 0;                                  /* initialize byte index */
    for (j=0;j<NPERMAP;j++) {               /* for each sensor, do */
      count = 0;                            /* initialize adjacency count */
      for (k=0;k<NPRUNTT;k++) {             /* do req no. times per sensor */
        for (l=0;l<8;l++) {                 /* for each bit, do */
          temp = q->sadajc[i];              /* establish working storage */
          temp =& (~(*mask[l]));            /* mask all but target bit */
          if (temp != 0)                    /* if adjacency bit on */
            count =+ 1;                     /* increment count */
        }
        i =+ 1;                             /* increment index */
      }
      if (count == 1)                       /* if a boundary sensor */
        return (j);                         /* return sensor index */
    }
    printf("no boundary sensors for array %d0,(q->aidback)->araynum);
      /* print error message */
    return (-1);                            /* and return error code */
}
```

```
/* This module contains operations necessary to complete the insertion
   of an array and operations performed on array nodes */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine creates a new array when the user needs it */
makarry () {
  int araynbr, i, fildnbr, rtnval, nstp, goon;
  char ans;
  struct fildid *pointr, *chkflst ();
  struct araynod *new;
  nstp = TRUE;                               /* set loop condition to no stop */
  while (nstp) {                             /* while not stop, do */
    printf("does field in which array will be located0); /* print user */
    printf("already exist ? (y or n)");                  /* instructions */
    ans = getans ();                         /* get user response */
    if (ans == 'n') {                        /* if no */
      fildnbr = makfild ();                  /* create a new array if possible */
      if (fildnbr != NIL) {                  /* if field was created */
        pointr = chkflst (fildnbr);          /* get pointer to field */
        nstp = FALSE;                        /* set loop stopping condition */
        printf("new field ID = %d0,fildnbr); /* give user new ID */
      }
      else {                                 /* otherwise */
        printf("do you still wish to insert ? (y or n)"); /* ask user */
        ans = getans ();                     /* get user response */
        if (ans == 'n')                      /* if no */
          return (TRMINAT);                 /* return with termination value */
      }
    }
    else {                                   /* otherwise */
      goon = TRUE;                           /* set loop condition to no stop */
      while (goon) {                         /* while not stop, do */
        fildnbr = getidnm ();                /* get ID number from user */
```

```
      if ((pointr = chkflst (fildnbr)) == NIL) { /* if bad ID */
        printf("no field associated with0); /* print error msg */
        printf("ID number given...0);
        printf("do you still want to insert ? (y or n)"); /* user instr */
        ans = getans ();                    /* get user response */
        if (ans == 'n')                     /* if no */
          return (TRMINAT);                 /* return termination value */
        else                                /* otherwise */
          goon = FALSE;                     /* set stopping condition */
      }
      else {                                /* otherwise, good ID number */
        goon = FALSE;                       /* set loop stopping condition */
        nstp = FALSE;
      }
    }
  }
}
new = (struct araynod *) calloc (1,(sizeof(struct araynod)));
  /* allocate storage for new array node */
rtnval = instand (new, pointr);     /* insert into array list in */
  /* proper position, link to field, increment number of arrays in */
  /* field (or return error) */
if (rtnval == ERROR) {                      /* if error */
  printf("cannot create an array for that field0); /* print error */
  printf("limit already reached...0);               /* message */
  cfree (new);                              /* deallocate storage */
  araynbr = NIL;                            /* set error condition */
}
else {                                      /* otherwise */
  printf("new array being created0); /* inform user */
  araynbr = getidnm ();                     /* get ID number from user */
  new->numsen = 0;                            /* set default conditions */
  for (i=0;i<NBYTES;i++)                    /* initialize bit map */
    new->sadajc[i] = 0;
  for (i=0;i<NTTYP;i++) {                   /* initialize velocity lists */
    new->vmin[i] = mintbl[i];
```

```
      new->vmax[i] = maxtbl[i];
      new->vav[i] = avtbl[i];
    }
    initsnl (new);                          /* initialize sensor list */
    araynbr = gtarval (new, araynbr); /* get array characteristics */
      /* from user, insert into ID list */
    if ((new->parnt)->numary > 1)          /* check for adjacency */
      aadjcnt (new, pointr);               /* if so, get from user */
  }
  return (araynbr);                        /* return to point of call */
}

/* This routine inserts a new array into the parent field array list
   in its proper position, links the new array to the parent field,
   increments the number of arrays counter in the field, or returns
   an error code if the insert was not possible */
instand (new, fieldlp)
  struct araynod *new;
  struct fildid *fieldlp; {
  struct araynod *r, *s;
  int rtnval;
  new->parnt = fieldlp->ptr;               /* link to parent node */
  if (((fieldlp->ptr)->numary) == NPERMAP) /* if parent already full */
    rtnval = ERROR;                        /* set error condition */
  else {                                   /* otherwise */
    rtnval = VALID;                        /* insertion is possible */
    r = (fieldlp->ptr)->header;            /* initialize search pointers */
    s = r->rt;
    while ((r->abmapno) == ((s->abmapno)-1)) { /* if insrt pos not fnd */
      r = s;                               /* advance pointers */
      s = r->rt;
    }
    insrt5 (new, s);                       /* insert node in the list */
    new->abmapno = r->abmapno + 1;         /* set bit map index */
    (fieldlp->ptr)->numary =+ 1;           /* increment no of arrays cntr */
  }
```

```
  return (rtnval);                          /* return status indicator */
}

/* This routine initializes a new sensor list given a pointer to the
   parent array node */
initsnl (new)
  struct araynod *new; {
  struct sensnod *first, *last;
  first = (struct sensnod *) calloc (1,(sizeof(struct sensnod)));
    /* allocate storage for the header node */
  last = (struct sensnod *) calloc (1,(sizeof(struct sensnod)));
    /* allocate storage for the sentinel node */
  new->headr = first;                       /* set parent links */
  new->sntinl = last;
  first->prev = NIL;                        /* establish list */
  first->nx = last;
  last->prev = first;
  last->nx = NIL;
  first->parent = new;                      /* link to parent */
  last->parent = new;
  first->type = NONE;                       /* designate as special nodes */
  last->type = NONE;
  first->sbmapno = NIL;
  last->sbmapno = NPERMAP;
  return;                                   /* return to point of call */
}

/* This routine checks to see if a newly created array is adjacent to
   any other already existent arrays, given a pointer to the new array
   and the parent field node */
aadjcnt (new, p)
  struct araynod *new;
  struct fildid *p; {
  int nstp, n, j, idno;
  char ans;
  struct arayid *ptr, *chkalst ();
```

```
  printf("is this array adjacent to any already0); /* print user */
  printf("created arrays ? (y or n)");             /* instructions */
  ans = getans ();                                 /* get user response */
  if (ans == 'y') {                                /* if yes */
    nstp = TRUE;                                   /* set loop condition to no stop */
    while (nstp) {                                 /* while not stop, do */
      printf("enter number of arrays the0); /* print user instructions */
      printf("new array is adjacent to");
      n = getint ();                               /* get user response */
      if ((n < 1) || (n > NPERMAP - 1)) /* if out of bounds */
        printf("number out of bounds0);/* print error message */
      else                                         /* otherwise */
        nstp = FALSE;                              /* good value */
    }
    for (j=0;j<n;j++) {                            /* for the number given, do */
      nstp = TRUE;                                 /* set loop condition to no stop */
      while (nstp) {                               /* while not stop, do */
        printf("for array %d adjacent to the new array0,j+1); /* print */
          /* user instructions */
        idno = getidnm ();                         /* get ID number */
        if((ptr = chkalst (idno)) == NIL) { /* if no associated array */
          printf("no array associated with0); /* print error msg */
          printf("ID number given0);
        }
        else                                       /* otherwise */
          nstp = FALSE;                            /* good input */
      }
      stabmap (new->abmapno,((ptr->pntr)->abmapno),p); /* set bit map */
    }
  }
  return;                                          /* return to point of call */
}

/* This routine checks to see if an array exists given an array number
   from the user.  If it exists it returns a pointer to the ID list node */
struct arayid *chkalst (id)
```

```
  int id; {
  struct arayid *p;
  aidtl->araynum = id;                    /* set stopping condition */
  p = aidhdr->next;                       /* initialize search pointer */
  while ((p->araynum) != id)              /* while not found, do */
    p = p->next;                          /* advance pointer */
  if (p == aidtl)                         /* if not found */
    return (NIL);                         /* return error condition */
  else                                    /* otherwise */
    return (p);                           /* return pointer to ID node */
}

/* This routine sets the array list bit map given the two indices and
   a pointer to the ID of the parent node */
stabmap (i, j, p)
  int i, j;
  struct fildid *p; {
  int bytindx, mskindx, temp, k;
  for (k=0;k<2;k++) {                     /* do twice */
    bytindx = NPRUNIT * i + j/8;          /* calculate byte index */
    mskindx = j % 8;                      /* calculate offset */
    (p->ptr)->aradajc[bytindx] =| mask[mskindx]; /* set bit map */
    temp = i;                             /* now reverse indices */
    i = j;
    j = temp;                             /* and repeat the process */
  }
  return;                                 /* return to point of call */
}

/* This routine gets array characteristics from the user, inserts the
   array ID number into the array ID list, and links the ID list node
   with the array node */
gtarval (new, araynbr)
  struct araynod *new;
  int araynbr; {
  char ans;
```

```
int val, nstp;
struct arayid *p;
nstp = TRUE;                              /* set loop condition to no stop */
p = (struct arayid *) calloc (1,(sizeof(struct arayid)));
  /* allocate storage for new ID node */
p->araynum = araynbr;                     /* initialize ID */
while ((val = insrt6 (p,aidhdr,aidtl)) != VALID) { /* while no insert */
  printf("the array ID number already used0); /* print error msg */
  printf("enter another0);                /* request another ID */
  p->araynum = getidnm ();                /* get user response */
}
p->pntr = new;                            /* link ID to node */
new->aidback = p;                         /* link node to ID */
while (nstp) {                            /* while not stop, do */
  printf("what type array is this - 0); /* ask for type */
  printf("t(rail  or  a(rea  ?0);
  ans = getans ();                        /* get user response */
  switch (ans) {                          /* case stmt for answer */
    case 't':
      new->typ = 10;                      /* type is trail */
      nstp = FALSE;                       /* set stopping condition */
      break;
    case 'a':
      new->typ = 20;                      /* type is array */
      nstp = FALSE;                       /* set stopping condition */
      break;
    default:
      printf("type not recognized0);/* print error message */
      break;
  }
}
printf("is the new array a boundary array ? (y or n)"); /* user instr */
ans = getans ();                           /* get user response */
if (ans == 'y') {                          /* if yes */
  new->typ =+ 1;                           /* set boundary indicator */
  printf("as a reminder, if the addition of this boundary array0);
```

```
    printf("causes other array characteristics to change you must0);
    printf("change them...0);          /* print user reminder */
  }
  printf("target velocity tables have been loaded0); /* user instr */
  printf("with the default values.  Changes required ? (y or n)");
  ans = getans ();                          /* get user response */
  if (ans =='y') {                          /* if yes */
    nstp = TRUE;                            /* set loop condition to no stop */
    while (nstp) {                          /* while not stop, do */
      printf("options:  a(verage vel chg   v(el max chg0); /* prnt user */
      printf("          m(in vel chg       q(uit0); /* instructions */
      printf("choose one...");
      ans = getans ();                      /* get user response */
      switch (ans) {                        /* case stmt for answer */
        case 'a':
          avchg (new);                      /* change to av vel table */
          break;
        case 'v':
          maxchg (new);                     /* change to max vel table */
          break;
        case 'm':
          minchg (new);                     /* change to min vel table */
          break;
        case 'q':
          nstp = FALSE;                     /* set stopping condition */
          break;
        default:
          printf("command not recognized0); /* print error msg */
          break;
      }
    }
  }
  return (p->araynum);                      /* return valid array ID */
}

/* This routine allows the user to make changes in the target average
```

```
   velocity table for an array */
avchg (new)
  struct araynod *new; {
  int i, nstp;
  nstp = TRUE;                              /* set loop condition to no stop */
  printf("changes to average velocity are made by type0); /* user instr */
  printf("and may be made until an invalid type number is input0);
  while (nstp) {                            /* while not stop, do */
    printf("enter target type");            /* user instructions */
    i = getint ();                          /* get user response */
    if ((i < 1) || (i > NTTYP))             /* if type out of limits */
      nstp = FALSE;                         /* set stopping condition */
    else {                                  /* otherwise */
      printf("average velocity for type %d0,i); /* user instr */
      printf("is currently %f0,new->vav[i-1]);
      printf("enter new value (m/sec)");    /* get new value */
      new->vav[i-1] = getfloat ();          /* store */
      printf("new value is %f0,new->vav[i-1]); /* echo to user */
    }
  }
  return;                                   /* return to point of call */
}

/* This routine allows the user to make changes in the target maximum
   velocity table for an array */
maxchg (new)
  struct araynod *new; {
  int i, nstp;
  nstp = TRUE;                              /* set loop condition to no stop */
  printf("changes to maximum velocity are made by type0); /* user instr */
  printf("and may be made until an invalid type number is input0);
  while (nstp) {                            /* while not stop, do */
    printf("enter target type");            /* user instructions */
    i = getint ();                          /* get user response */
    if ((i < 1) || (i > NTTYP))             /* if type out of limits */
      nstp = FALSE;                         /* set stopping condition */
```

```
    else {                                  /* otherwise */
      printf("maximum velocity for type %d0,i); /* user instr */
      printf("is currently %f0,new->vav[i-1]);
      printf("enter new value (m/sec)"); /* get new value */
      new->vmax[i-1] = gtfloat ();       /* store */
      printf("new value is %f0,new->vmax[i-1]); /* echo to user */
    }
  }
  return;                                   /* return to point of call */
}

/* This routine allows the user to make changes in the target minimum
   velocity table for an array */
minchg (new)
  struct araynod *new; {
  int i, nstp;
  nstp = TRUE;                              /* set loop condition to no stop */
  printf("changes to minimum velocity are made by type0); /* user instr */
  printf("and may be made until an invalid type number is input0);
  while (nstp) {                            /* while not stop, do */
    printf("enter target type");            /* user instructions */
    i = getint ();                          /* get user response */
    if ((i < 1) || (i > NTTYP))             /* if type out of limits */
      nstp = FALSE;                         /* set stopping condition */
    else {                                  /* otherwise */
      printf("minimum velocity for type %d0,i); /* user instr */
      printf("is currently %f0,new->vmin[i-1]);
      printf("enter new value (m/sec)"); /* get new value */
      new->vmin[i-1] = gtfloat ();       /* store */
      printf("new value is %f0,new->vmin[i-1]); /* echo to user */
    }
  }
  return;                                   /* return to point of call */
}

/* This routine gets and returns the minimum target velocity given a
```

```
   pointer to the array node */
float getmin (q)
  struct araynod *q; {
  float minval;
  int i;
  minval = 100000.0;                    /* initialize minimum value */
  for (i=0;i<NTTYP;i++)                 /* for all target types, do */
    if (q->vmin[i] < minval)            /* if a new minimum value */
      minval = q->vmin[i];              /* save new value */
  return (minval);                      /* return value to point of call */
}
```

```
/* This module performs operations necessary to complete the insertion of
   a field */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine creates a new field when the user needs it */
makfild () {
  int fildnbr, i, rtnval;
  struct fildnod *new;
  new = (struct fildnod *) calloc (1,(sizeof(struct fildnod)));
    /* allocate storage for the new node */
  rtnval = instfnd (new);                   /* insert into field list in
    proper position, link to root, increment number of fields in root,
    or return error */
  if (rtnval == ERROR) {                     /* if no insert */
    fildnbr = NIL;                           /* set error condition */
    cfree (new);                             /* deallocate storage */
    printf("cannot create a new field0); /* print error msg */
    printf("limit already reached0);
  }
  else {                                     /* otherwise */
    printf("new field being created0); /* notice to user */
    fildnbr = getidnm ();                    /* get ID number */
    new->numary = 0;                         /* set default conditions */
    for (i=0;i<NBYTES;i++)                   /* initialize bit map */
      new->aradajc[i] = 0;
    initanl (new);                           /* initialize array list */
    fildnbr = gtflval (new,fildnbr);    /* get field characteristics */
    if ((new->rootpar)->numfld > 1)          /* if possibility of adjacency */
      fadjcnt (new);                         /* query user */
  }
  return (fildnbr);                          /* return valid field number */
}
```

```
/* This routine initializes a new array list given a pointer to the
   parent field node */
initanl (new)
  struct fildnod *new; {
  struct araynod *first, *last;
  first = (struct araynod *) calloc (1,(sizeof(struct araynod)));
    /* allocate storage for the header node */
  last = (struct araynod *) calloc (1,(sizeof(struct araynod)));
    /* allocate storage for the sentinel node */
  new->header = first;                          /* set parent links */
  new->sntnl = last;
  first->lf = NIL;                              /* establish list */
  first->rt = last;
  last->lf = first;
  last->rt = NIL;
  first->parnt = new;                           /* link to parent */
  last->parnt = new;
  first->typ = NONE;                            /* designate as special nodes */
  last->typ = NONE;
  first->abmapno = NIL;
  last->abmapno = NPERMAP;
  return;                                       /* return to point of call */
}

/* This routine gets field characteristics from the user, inserts the
   field ID number into the field ID list, and links the ID list node
   with the field node */
gtflval (new, fildnbr)
  struct fildnod *new;
  int fildnbr; {
  char ans;
  int val;
  struct fildid *p;
  p = (struct fildid *) calloc (1,(sizeof(struct fildid)));
    /* allocate storage for new ID node */
  p->fildnum = fildnbr;                         /* initialize ID */
```

```
  while ((val = insrt8 (p,flidhdr,flidtl)) != VALID) { /* while no insrt */
    printf("the field ID number already used0);  /* print error msg */
    printf("enter another0);                     /* request another ID */
    p->fildnum = getidnm ();                     /* get user response */
  }
  p->ptr = new;                                  /* link ID to node */
  new->fidback = p;                              /* link node to ID */
  printf("is the new field a boundary field ? (y or n)"); /* user instr */
  ans = getans ();                               /* get user response */
  if (ans == 'y') {                              /* if yes */
    new->feldtyp = 1;                            /* set boundary indicator */
    printf("as a reminder, if the addition of this boundary field0);
    printf("causes other field characteristics to change you must0);
    printf("change them...0);                  /* print user reminder */
  }
  else                                           /* otherwise */
    new->feldtyp = 0;                            /* set to not boundary */
  return (p->fildnum);                           /* return valid field ID */
}

/* This routine inserts a new field node into the field list of the root
   in its proper position, links the new field to the root node,
   increments the number of fields counter in the root, or returns
   an error code if the insert was not possible */
instfnd (new)
```

```
struct fldhdr *r, *s;
int rtnval;
new->rootpar = rootptr;                    /* link to parent node */
if ((rootptr->numfld) == NPERMAP)         /* if parent already full */
  rtnval = ERROR;                          /* set error condition */
else {                                     /* otherwise */
  rtnval = VALID;                          /* insertion is possible */
  r = rootptr->fldhdr;                     /* initialize search pointers */
  s = r->fnext;
  while ((r->fbmapno) == ((s->fbmapno)-1)) { /* if insrt pos not fnd */
```

```
      r = s;                                  /* advance pointers */
      s = r->fnext;
    }
    insrt7 (new, s);                         /* insert node in the list */
    new->fbmapno = r->fbmapno + 1;           /* set bit map index */
    rootptr->numfld =+ 1;                    /* increment no of arrays cntr */
  }
  return (rtnval);                           /* return status indicator */
}

/* This routine checks to see if a newly created field is adjacent to
   any other already existent fields, given a pointer to the new field */
fadjcnt (new)
  struct fildnod *new; {
  int nstp, n, j, idno;
  char ans;
  struct fildid *pntr, *chkflst ();
  printf("is this field adjacent to any already0); /* print user */
  printf("created fields ? (y or n)");             /* instructions */
  ans = getans ();                           /* get user response */
  if (ans == 'y') {                          /* if yes */
    nstp = TRUE;                             /* set loop condition to no stop */
    while (nstp) {                           /* while not stop, do */
      printf("enter number of fields the0); /* print user instructions */
      printf("new field is adjacent to");
      n = getint ();                         /* get user response */
      if ((n < 1) || (n > NPERMAP - 1)) /* if out of bounds */
        printf("number out of bounds0);/* print error message */
      else                                   /* otherwise */
        nstp = FALSE;                        /* good value */
    }
    for (j=0;j<n;j++) {                      /* for the number given, do */
      nstp = TRUE;                           /* set loop condition to no stop */
      while (nstp) {                         /* while not stop, do */
        printf("for field %d adjacent to the new field0,j+1); /* print */
          /* user instructions */
```

```
        idno = getidnm ();                  /* get ID number */
        if((pntr = chkflst (idno)) == NIL) { /* if no associated field */
          printf("no field associated with0); /* print error msg */
          printf("ID number given0);
        }
        else                                /* otherwise */
          nstp = FALSE;                     /* good input */
      }
      stfbmap (new->fbmapno,((pntr->ptr)->fbmapno)); /* set bit map */
    }
  }
  return;                                   /* return to point of call */
}

/* This routine checks to see if a field exists given a field number
   from the user.  If it exists it returns a pointer to the ID list node */
struct fildid *chkflst (id)
  int id; {
  struct fildid *p;
  flidtl->fildnum = id;                     /* set stopping condition */
  p = flidhdr->follow;                      /* initialize search pointer */
  while ((p->fildnum) != id)                /* while not found, do */
    p = p->follow;                          /* advance pointer */
  if (p == flidtl)                          /* if not found */
    return (NIL);                           /* return error condition */
  else                                      /* otherwise */
    return (p);                             /* return pointer to ID node */
}

/* This routine sets the field list bit map given the two indices */
stfbmap (i, j)
  int i, j; {
  int bytindx, mskindx, temp, k;
  for (k=0;k<2;k++) {                       /* do twice */
    bytindx = NPRUNIT * i + j/8;            /* calculate byte index */
    mskindx = j % 8;                        /* calculate offset */
```

```
    rootptr->fadajc[bytindx] =| mask[mskindx];    /* set bit map */
    temp = i;                                     /* now reverse indices */
    i = j;
    j = temp;                                     /* and repeat the process */
  }
  return;                                         /* return to point of call */
}
```

```
/* This module contains routines that operate on filtered data lists */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine creates and inserts a new filtered sensor data list
   list node, given a pointer to the parent sensor */
struct sdatafl *cfsdnln (q)
  struct sensnod *q; {
  struct sdatafl *p;
  p = (struct sdatafl *) calloc (1,(sizeof(struct sdatafl)));
    /* allocate storage for the new node */
  insrt22 (p,q);                             /* insert the node in the list */
  intfsdl (p);                               /* create assoc filtrd data list */
  return (p);                                /* return pointer to new node */
}

/* This routine creates a new filtered data list given a pointer to the
   parent list node */
intfsdl (new)
  struct sdatafl *new; {
  struct sdataf *first, *last;
  first = (struct sdataf *) calloc (1,(sizeof(struct sdataf)));
    /* allocate storage for the header node */
  last = (struct sdataf *) calloc (1,(sizeof(struct sdataf)));
    /* allocate storage for the sentinel node */
  new->dfhead = first;                       /* set parent links */
  new->dfsent = last;
  first->lft = NIL;                          /* establish list */
  first->rght = last;
  last->lft = first;
  last->rght = NIL;
  first->pnt = new;                          /* link to parent */
  last->pnt = new;
  /* designate as special nodes */
```

```
  return;                                   /* return to point of call */
}

/* This routine searches for a filtered data list node list node of a
   given type, given a pointer to the parent sensor node.  It returns
   a pointer to the node or a flag indicating that the node was not
   found */
struct sdataf1 *sfsdnln (q, type)
  struct sensnod *q;
  int type; {
  struct sdataf1 *p;
  p = (q->hd)->fdlnx;                       /* set pointer to 1st node */
  (q->tl)->fdltyp = type;                   /* set stopping condition */
  while (p->fdltyp != type)                 /* while not found, do */
    p = p->fdlnx;                           /* advance pointer */
  if (p == q->tl)                           /* if not found */
    return (NIL);                           /* return not found flag */
  else                                      /* otherwise */
    return (p);                             /* return pointer to node */
}

/* This routine creates and inserts a new filtered data list node,
   given a pointer to the parent list */
struct sdataf *cfsdn (q)
  struct sdataf1 *q; {
  struct sdataf *p;
  p = (struct sdataf *) calloc (1,(sizeof(struct sdataf)));
    /* allocate storage for the new node */
  insrt2 (p, q);                    /* insert the new node in the list */
  return (p);                       /* return the pointer to the new node */
}

/* This routine, given a pointer to a filtered data node occuring in
   the present time window, checks the valid set list to see if a valid
   set already exists for the activations (extension from last time period).
   If so, it returns a pointer to the valid set; otherwise it returns a
```

```
   not found flag */
struct vset *vstofdn (q)
  struct sdataf *q; {
  struct sensnod *s;
  struct sdataf *t;
  struct vset *p;
  int keepon, i;
  keepon = TRUE;                          /* set loop condition to no stop */
  s = ((q->pnt)->dlprnt);                 /* point to sensor */
  p = (s->hdr)->foll;                     /* point to 1st valid set node */
  while ((p != s->sentl) && (keepon)) {   /* while node not found, do */
    t = p->lst;
    for (i=0;i<(p->numact-1);i++)         /* move thru fil data list */
      t = t->rght;
    if (t == q->lft)                      /* if nodes are adjacent */
      if (q->tmact <= (t->tmact + s->hs + MASMEN)) /* if continuous activ */
        keepon = FALSE;                   /* set stopping condition */
    p = p->foll;                          /* advance pointer */
  }
  if (keepon)                             /* if node not found */
    return (NIL);                         /* return not found flag */
  else                                    /* otherwise */
    return (p->prevs);                    /* return pointer to valid set */
}

/* This routine gets the number of activations to add onto a valid set
   given a pointer to the first activation in the valid set */
gtnmact (q)
  struct sdataf *q; {
  int cntr;
  cntr = 0;                               /* initialize node counter */
  while (q->tmact < datatim)              /* skip all counted previously */
    q = q->rght;
  while (q != (q->pnt)->dfsent) {         /* while not thru the list, do */
    cntr =+ 1;                            /* increment the counter */
    q = q->rght;                          /* advance the pointer */
```

```
  }
  return (cntr);                          /* return the node count */
}

/* This routine searches the filtered data list for the first node that
   created in the present time window, given a pointer to the parent
   filtered data list list node.  If a node is found a pointer to that
   node is returned; otherwise a not found flag is returned */
struct sdataf *srfldl (q)
  struct sdatafl *q; {
  struct sdataf *p;
  p = (q->dfhead)->rght;                  /* point to 1st fil data node */
  (q->dfsent)->tmact = datatim;           /* set stopping condition */
  while (p->tmact < datatim)              /* while node not found */
    p = p->rght;                          /* advance pointer */
  if (p == q->dfsent)                     /* if not found */
    return (NIL);                         /* return not found flag */
  else                                    /* otherwise */
    return (p);                           /* return pointer to node */
}
```

```
/* This module contains routines that operate on valid set lists */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine checks to see if a valid set will be complete or
   incomplete given a pointer to the 1st filtered data node of the
   valid set */
vscompl (p)
  struct sdataf *p; {
  struct sdataf *q;
  q = p->rght;                               /* set pointer to next node */
  while (q != (p->pnt)->dfsent) {            /* while not thru list, do */
    p = q;                                   /* advance pointers */
    q = p->rght;
  }
  if ((p->tmact + (((p->pnt)->dlprnt)->hs) + MASMEN) < systime) /* compl? */
    return (YES);                            /* return completed */
  else                                       /* otherwise */
    return (NO);                             /* return uncompleted */
}

/* This routine creates a new valid set node, inserts it in the valid
   set list, and creates an event hypothesis list for the new node, given
   a pointer to the parent sensor */
struct vset *crtvset (q)
  struct sensnod *q; {
  struct vset *p;
  p = (struct vset *) calloc (1,(sizeof(struct vset)));
    /* allocate storage for the new node */
  insrt9 (p, q);                                 /* insert the new node */
  crtehl (p);                                    /* create evnt hypoth list */
  return (p);                                    /* return pointer to node */
}
```

```
/* This routine calculates and returns the centroid time of a valid
   set given a pointer to the first filtered data list node in the valid
   set */
float gtcntim (p)
  struct sdataf *p; {
  struct sdataf *q, *r;
  float valu;
  q = p;                                        /* initialize pointers */
  r = q->rght;
  while (r != (p->pnt)->dfsent) {               /* while not thru list, do */
    q = r;                                      /* advance pointers */
    r = q->rght;
  }
  valu = .5 * (p->tmact + q->tmact);            /* calculate centroid time */
  return (valu);                                /* return centroid value */
}

/* This routine creates a new event hypothesis list given a pointer to
   the parent valid set */
crtehl (p)
  struct vset *p; {
  struct evntl *first, *last;
  first = (struct evntl *) calloc (1,(sizeof(struct evntl)));
    /* allocate storage for the new header node */
  last = (struct evntl *) calloc (1,(sizeof(struct evntl)));
    /* allocate storage for the new sentinel node */
  first->vlower = p;                                /* link to parent */
  last->vlower = p;
  p->higher = first;                                /* both ways */
  p->eltail = last;
  first->follw = last;                              /* create list */
  first->prvious = NIL;
  last->follw = NIL;
  last->prvious = first;
  first->link = NIL;                                /* designate as special nodes */
  last->link = NIL;
```

```
  return;                                   /* return to point of call */
}

/* This routine eliminates a valid set when required, given a pointer to
   the node to be removed */
rmvset (t)
  struct vset *t; {
  struct evntl *n, *temp;
  n = (t->higher)->follw;                   /* point to 1st list node, if any */
  while (n != t->eltail) {                  /* while not thru list, do */
    temp = n;
    n = n->follw;                           /* delete node from the list */
    dlete10 (temp);
    cfree (temp);                           /* free the storage */
  }
  cfree (t->eltail);                        /* free sentinel node storage */
  cfree (t->higher);                        /* free header node storage */
  dlete9 (t);                               /* delete valid set from list */
  cfree (t);                                /* free the storage */
  return;                                   /* return to point of call */
}

/* This recursive routine checks for time ordered groups of valid sets
   to form event hypotheses.  If it finds a group meeting the minimum
   requirements, it causes the appropiate hypothesis to be generated */
search (k,m,sptr,ptr,snsrcnt,g)
  int k, m, snsrcnt;
  struct sensnod *sptr[];
  struct vset *ptr[];
  struct araynod *g; {
  int count, counter, nostop, l;
  count = 0;                                /* set hypoth total count */
  counter = 0;                              /* set by node hypoth count */
  while (ptr[k] != sptr[k]->sentl) {        /* while not thru list, do */
    nostop = TRUE;                          /* set loop condition */
    while ((k < (snsrcnt - 1)) && (nostop)) { /* while not at bottom lvl */
```

```
      if ((k == m) || (ptr[k]->centrd >= ptr[k-1]->centrd)) {
        /* if at top level or a time ordered pair */
        l = k + 1;                              /* search at next level */
        counter = search (l,m,sptr,ptr,snsrcnt,q);
        count =+ counter;                       /* increment total count */
      }
      else                                      /* otherwise */
        nostop = FALSE;                         /* break at intermed level */
      if ((count == 0) && ((k-m) >= (NUMT-1)) && (nostop)) {
        /* if no hypotheses from lower level and one can be formed */
        csevhyp (m,k,q,sptr,ptr);           /* create hyp and set params */
        count =+ 1;                             /* increment hypoth count */
      }
      nostop = FALSE;                           /* set stopping condition */
      counter = 0;                              /* reset hypoth count */
    }
    if ((ptr[k]->centrd >= ptr[k-1]->centrd) && (nostop)) {
      /* if all levels are time ordered */
      csevhyp (m,k,q,sptr,ptr);             /* create hyp and set params */
      count =+ 1;                               /* increment hypoth count */
    }
    ptr[k] = ptr[k]->foll;                      /* advance pointer */
  }
  ptr[k] = (sptr[k]->hdr)->foll;                /* reset pntr to 1st vs node */
  return (count);                               /* ret hypoth count for lvl */
}
```

```
/* This module contains routines that operate on event hypothesis nodes */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine checks to see if two event hypothesis nodes are duplicates
   of each other, given pointers to the two nodes.  It returns a condition
   code indicating the result of the check */
dupehy (p, q)
  struct evnt *p, *q; {
  struct vsetl *r, *s;
  struct velist *t, *u;
  int indcat;
  if (p->numvset != q->numvset)              /* if no. of vs's unequal */
    return (NO);                             /* return not equal flag */
  else {                                     /* otherwise */
    r = (p->lowr)->lfoll;                    /* set pointers to each */
    s = (q->lowr)->lfoll;                    /* valid set list */
    while ((r != p->vstail) && (r->vslink == s->vslink)) {
      /* continue until a stopping condition occurs */
      r = r->lfoll;                          /* advance pointers */
      s = s->lfoll;
    }
    if (r != p->vstail)                       /* not thru list */
      return (NO);                           /* return not equal flag */
    else {                                   /* otherwise */
      indcat = YES;                          /* set equivalancy indicat */
      t = (p->vhdr)->vnext;                  /* set pointers to each */
      u = (q->vhdr)->vnext;                  /* velocity list */
      while ((t != p->vsent) && (u != q->vsent)) { /* while not thru lsts */
        if ((t->sens1 != u->sens1) || (t->sens2 != u->sens2)
          || (t->vel != u->vel))             /* if nodes not equal */
          indcat = NO;                       /* reset indicator */
        t = t->vnext;                        /* advance pointers */
        u = u->vnext;
```

```
      }
      if ((t == p->vsent) && (u == q->vsent)) /* if lists equal length */
        return (indcat);                      /* return appropiate code */
      else                                    /* otherwise */
        return (NO);                          /* return not equal flag */
    }
  }
}

/* This routine checks to see if an event hypothesis node is a
   subextension of the other, given pointers to the two nodes.  It returns
   a condition code indicating the result of the check */
sextehy (p, q)
  struct evnt *p, *q; {
  struct vsetl *r, *s;
  struct velist *t, *u;
  int indcat, difvel;
  if (p->numvset != q->numvset)               /* if no. of vs's unequal */
    return (NO);                              /* return not subext flag */
  else {                                      /* otherwise */
    r = (p->lowr)->lfoll;                     /* set pointers to each */
    s = (q->lowr)->lfoll;                     /* valid set list */
    while ((r != p->vstail) && (r->vslink == s->vslink)) {
      /* continue until a stopping condition occurs */
      r = r->lfoll;                           /* advance pointers */
      s = s->lfoll;
    }
    if (r != p->vstail)                        /* not thru list */
      return (NO);                             /* return not subext flag */
    else {                                     /* otherwise */
      indcat = YES;                            /* set equivalancy indicat */
      difvel = NO;                             /* set subextension flag */
      t = (p->vhdr)->vnext;                    /* set pointers to each */
      u = (q->vhdr)->vnext;                    /* velocity list */
      while ((t != p->vsent) && (u != q->vsent)) { /* while not thru lsts */
        if ((t->sens1 != u->sens1) || (t->sens2 != u->sens2))
```

```
            /* if nodes not equal */
            indcat = NO;                              /* reset indicator */
          else                                        /* otherwise */
            if (t->vel != u->vel)                     /* if velocities not equal */
              difvel = YES;                           /* reset indicator */
          t = t->vnext;                               /* advance pointers */
          u = u->vnext;
        }
        if ((t == p->vsent) && (u == q->vsent)        /* if lists equal length */
          && (difvel == YES))                         /* and nodes different */
          return (indcat);                            /* return appropiate code */
        else                                          /* otherwise */
          return (NO);                                /* return not equal flag */
      }
    }
}

/* This routine checks to see if an event hypothesis node is a subset/
   subextension or a subset of the other given pointers to the two nodes.
   It returns a condition code indicating the result of the check */
ssetehy (p, q)
  struct evnt *p, *q; {
  struct vsetl *r, *s;
  r = (p->lowr)->lfoll;                          /* set pointers to each */
  s = (q->lowr)->lfoll;                          /* valid set list */
  while ((r != p->vstail) && (r->vslink == s->vslink)
    && (s != q->vstail)) {                       /* while no stopping cond */
    r = r->lfoll;                                /* advance pointers */
    s = s->lfoll;
  }
  if ((r == p->vstail) && (s != q->vstail))      /* if a subset */
    return (YES);                                /* return subset flag */
  else {                                         /* otherwise */
    r = (p->vstail)->lprev;                      /* set pntrs to last nodes */
    s = (q->vstail)->lprev;
    while ((r != p->lowr) && (s != q->lowr) &&
```

```
      (r->vslink == s->vslink)) {          /* while no stopping cond */
      r = r->lprev;                        /* regress pointers */
      s = s->lprev;
    }
    if ((r == p->lowr) && (s != q->lowr))  /* if a subset other way */
      return (YES);                        /* return subset indicator */
    else                                   /* otherwise */
      return (NO);                         /* return no subset */
  }
}

/* This routine forms an event hypothesis given the level numbers of
   the involved valid sets and a pointer to the parent array */
csevhyp (m,k,q,sptr,ptr)
  int m, k;
  struct araynod *q;
  struct sensnod *sptr[];
  struct vset *ptr[]; {
  struct vsetl *r;
  struct evnt *p, *crevhyp ();
  struct velist *s;
  struct evntl *t;
  int count, i, j, stype[NPERMAP], sclass[NPERMAP];
  float vlmax, vlmin, vlav, dist;
  double x, y;
  vlmax = 0.0;                             /* initialize computation var */
  vlmin = 9999.0;
  vlav = 0.0;
  count = 0;
  p = crevhyp ();                          /* create new event node */
  i = m;                                   /* initialize index */
  while (i < k) {                         /* for all sensor pairs */
    j = i + 1;                             /* set second index */
    while (j <= k) {                       /* while not thru list */
      s = (struct velist *) calloc (1,(sizeof(struct velist)));
        /* allocate storage for new velocity node */
```

```
    s->sens1 = (sptr[i]->sidback)->idno;  /* set 1st sensor id */
    s->sens2 = (sptr[j]->sidback)->idno;  /* set 2nd sensor id */
    x = (double)((sptr[i]->slocx - sptr[j]->slocx) * 10.0);
    y = (double)((sptr[i]->slocy - sptr[j]->slocy) * 10.0);
      /* calculate distance between sensors */
    dist = ((float)(sqrt(pow(x,((double)(2.0))) +
      pow(y,((double)(2.0))))));
      /* and target velocity between sensors */
    s->vel = dist/((float)fabs((double)(ptr[i]->centrd-ptr[j]->centrd)));
    count =+ 1;                           /* increment node count */
    vlav =+ s->vel;                       /* sum for average vel calc */
    if (s->vel < vlmin)                   /* if velocity less than min */
      vlmin = s->vel;                     /* save the value */
    if (s->vel > vlmax)                   /* if greater than max */
      vlmax = s->vel;                     /* save the value */
    insrt12 (s, p);                       /* insert the velocity node */
    j =+ 1;                               /* increment for next check */
  }
  i =+ 1;                                 /* increment for nx iteration */
}
p->velav = (vlav / count);                /* set average velocity */
p->velmax = vlmax;                        /* set maximum velocity */
p->velmin = vlmin;                        /* set minimum velocity */
p->numvset = k - m + 1;                   /* set no. of valid sets */
p->numcl = 0;                             /* set no. of classifications */
p->updtim = systime;                      /* set latest update time */
insrt11 (p, evtl);                        /* insert event itself */
i = m;                                    /* reset index */
for (j=0;j<(k-m+1);j++) {                 /* for all sensors, do */
  r = (struct vsetl *) calloc (1,(sizeof(struct vsetl)));
    /* allocate storage for new valid set list node */
  r->vslink = ptr[i];                     /* link to valid set */
  insrt13 (r, p);                         /* insert into the list */
  t = (struct evntl *) calloc (1,(sizeof(struct evntl)));
    /* allocate storage for the new event list node */
  t->link = p;                            /* link to event */
```

```
      insrt10 (t, ptr[i]);                          /* insert into the list */
      stype[j] = ((sptr[i]->type - ((sptr[i]->type/10000) * 10000))/1000);
        /* get sensor type */
      sclass[j] = (sptr[i]->type/10000);
       /* get sensor class */
      i =+ 1;                                       /* increment index */
    }
    if ((q->typ / 10) == 1)                         /* if a trail array */
      p->etyp = 0;                                  /* set event type */
    else                                            /* otherwise */
      p->etyp = 100;                                /* set to area type */
    i = 1;                                          /* reset index */
    while ((sclass[i] == sclass[i-1]) && (i++ != (k-m+1)));
      /* do classification check */
    if (i == (k-m+1))                               /* if list end reached */
      p->etyp =+ (sclass[0] * 10);                  /* set class */
    i = 1;                                          /* reset index */
    while ((stype[i] == stype[i-1]) && (i++ != (k-m+1)));
      /* do type check */
    if (i == (k-m+1))                               /* if list end reached */
      p->etyp =+ (stype[0] + 1);                    /* set type */
    return;                                         /* return to point of call */
}

/* This routine creates an event hypothesis and returns a pointer to the
   newly created node */
struct evnt *crevhyp () {
  struct evnt *p;
  p = (struct evnt *) calloc (1,(sizeof(struct evnt)));
    /* allocate storage for the new event hypothesis node */
  p->vhdr = (struct velist *) calloc (1,(sizeof(struct velist)));
    /* allocate storage for the velocity list header node */
  p->vsent = (struct velist *) calloc (1,(sizeof(struct velist)));
    /* allocate storage for the velocity list sentinel node */
  (p->vhdr)->vnext = p->vsent;                      /* form velocity list */
  (p->vhdr)->vlast = NIL;
```

```
  (p->vsent)->vnext = NIL;
  (p->vsent)->vlast = p->vhdr;
  p->hgher = (struct chypl *) calloc (1,(sizeof(struct chypl)));
    /* allocate storage for the class hypoth list header node */
  p->chtail = (struct chypl *) calloc (1,(sizeof(struct chypl)));
    /* allocate storage for the class hypoth list sentinel node */
  (p->hgher)->cnxt = p->chtail;                    /* form class hyp list */
  (p->hgher)->cprev = NIL;
  (p->chtail)->cnxt = NIL;
  (p->chtail)->cprev = p->hgher;
  p->lowr = (struct vsetl *) calloc (1,(sizeof(struct vsetl)));
    /* allocate storage for the valid set list header node */
  p->vstail = (struct vsetl *) calloc (1,(sizeof(struct vsetl)));
    /* allocate storage for the valid set list sentinel node */
  (p->lowr)->lfoll = p->vstail;                    /* form valid set list */
  (p->lowr)->lprev = NIL;
  (p->vstail)->lfoll = NIL;
  (p->vstail)->lprev = p->lowr;
  return (p);                                      /* return pointer */
}

/* This routine removes an event hypothesis node from the event hypothesis
   list given a pointer to the node */
remehyp (p)
  struct evnt *p; {
  struct evntl *u;
  struct vsetl *q, *t3;
  struct eventl *t;
  struct chypl *r, *t1;
  struct velist *s, *t2;
  int i;
  r = (p->hgher)->cnxt;                        /* point to 1st cl hyp lst node */
  for (i=0;i<p->numcl;i++) {                   /* do for all cl hyp list nodes */
    t = ((r->hlink)->clower)->elnext;          /* pnt to 1st node in ev hyp ls */
    while (t->elink != p)                      /* while not at own node */
      t = t->elnext;                           /* advance pointer */
```

```
    t->elink = NIL;                          /* mark for removal later */
    t1 = r;                                  /* set temporary pointer */
    r = r->cnxt;                             /* advance list pointer */
    dlete14 (t1);                            /* delete node from list */
    cfree (t1);                              /* and free storage */
  }
  cfree (p->hgher);                          /* remove header and */
  cfree (p->chtail);                         /* sentinel nodes too */
  s = (p->vhdr)->vnext;                      /* point to 1st vel list node */
  while (s != p->vsent) {                    /* while not thru list, do */
    t2 = s;                                  /* set temporary pointer */
    s = s->vnext;                            /* advance list pointer */
    dlete12 (t2);                            /* delete node from list */
    cfree (t2);                              /* and free storage */
  }
  cfree (p->vhdr);                           /* remove header and */
  cfree (p->vsent);                          /* sentinel nodes too */
  q = (p->lowr)->lfoll;                      /* point to 1st vs list node */
  for (i=0;i<p->numvset;i++) {               /* for all vs list nodes, do */
    u = ((q->vslink)->higher)->follw;        /* point to 1st ev hyp ls node */
    while (u->link != p)                     /* while not at own node */
      u = u->follw;                          /* advance pointer */
    dlete10 (u);                             /* delete node from the list */
    cfree (u);                               /* and free the storage */
    t3 = q;                                  /* set temporary pointer */
    q = q->lfoll;                            /* advance list pointer */
    dlete13 (t3);                            /* delete node from list */
    cfree (t3);                              /* and free the storage */
  }
  cfree (p->lowr);                           /* remove header and */
  cfree (p->vstail);                         /* sentinel nodes too */
  dlete11 (p);                               /* delete the hyp node itself */
  cfree (p);                                 /* and free the storage */
  return;                                    /* return to point of call */
}
```

```
/* This module contains routines that operate on classification
   hypothesis nodes */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine creates and sets the parameters of a classification
   hypothesis node, given pointers to the involved event hypothesis
   and the involved array, and the type of classification */
csclhyp (q, r, n)
  struct evnt *q;
  struct araynod *r;
  int n; {
  struct classh *p, *crclhyp ();
  struct chypl *s;
  struct eventl *t;
  p = crclhyp ();                            /* create new class hyp node */
  q->numcl =+ 1;                             /* incr event no. of class */
  s = (struct chypl *) calloc (1,(sizeof(struct chypl)));
    /* allocate storage for new class hypoth list node */
  s->hlink = p;                              /* establish link */
  insrt14 (s, q);                            /* and insert into list */
  stchtyp (p, q, r);                         /* set classification type */
  p->class = n;                              /* set classification code */
  p->cvel = q->velav;                        /* set classification vel */
  stchdir (p, q, r);                         /* set classification dir */
  stchln (p, q, r);                          /* set classification length */
  stchnum (p, q, r);                         /* set classification number */
  stchloc (p, q, r);                         /* set classification locn */
  p->ceftim = systime;                       /* set effective calc time */
  p->cupd = systime;                         /* set last update time */
  stchcf (q, r);                             /* set certainty factor */
  t = (struct eventl *) calloc (1,(sizeof(struct eventl)));
    /* allocate storage for new event list node */
  t->elink = q;                              /* establish link */
```

```
  insrt16 (t, p);                          /* insert into the list */
  insrt15 (p, cltl);                       /* insert the class nod itself */
  return;                                  /* return to point of call */
}

/* This routine sets the type of the classification hypothesis,
   given pointers to the classification hypothesis, the involved
   event, and the involved array */
stchtyp (p, q, r)
  struct classh *p;
  struct evnt *q;
  struct araynod *r; {
  /* option not implemented */
  return;                                  /* return to point of call */
}

/* This routine sets the target direction of the classification hypothesis,
   given pointers to the classification hypothesis, the involved
   event, and the involved array */
stchdir (p, q, r)
  struct classh *p;
  struct evnt *q;
  struct araynod *r; {
  /* option not implemented */
  return;                                  /* return to point of call */
}

/* This routine sets the target length of the classification hypothesis,
   given pointers to the classification hypothesis, the involved
   event, and the involved array */
stchln (p, q, r)
  struct classh *p;
  struct evnt *q;
  struct araynod *r; {
  /* option not implemented */
  return;                                  /* return to point of call */
```

```
}

/* This routine sets the target number of the classification hypothesis,
   given pointers to the classification hypothesis, the involved
   event, and the involved array */
stchnum (p, q, r)
  struct classh *p;
  struct evnt *q;
  struct araynod *r; {
  /* option not implemented */
  return;                                    /* return to point of call */
}

/* This routine sets the target location of the classification hypothesis,
   given pointers to the classification hypothesis, the involved
   event, and the involved array */
stchloc (p, q, r)
  struct classh *p;
  struct evnt *q;
  struct araynod *r; {
  /* option not implemented */
  return;                                    /* return to point of call */
}

/* This routine sets the certainty factors for all the classification
   hypotheses associated with an event hypothesis, given a pointer to
   that event hypothesis and a pointer to the involved array */
stchcf (q, r)
  struct evnt *q;
  struct araynod *r; {
  struct classh *ptr[NTTYP];
  struct chypl *s;
  int i;
  float ymin, temp, ybar[NTTYP], x;
  i = 0;                                      /* initialize index */
  s = (q->hgher)->cnxt;                      /* pnt to 1st cl hyp 1st nod */
```

```
  while (s != q->chtail) {                          /* while not thru list, do */
    ptr[i++] = s->hlink;                            /* set pointer to class hyp */
    s = s->cnxt;                                    /* advance pointer */
  }
  ymin = 99999.0;                                   /* initialize min value */
  for (i=0;i<q->numcl;i++) {                        /* for all classifications */
    temp = (float)fabs((double)(q->velav-r->vav[ptr[i]->class]));
      /* get difference between event velocity and type average velocity */
    ybar[i] = temp;                                 /* save value for later */
    if (temp < ymin)                                /* if new min value */
      ymin = temp;                                  /* save it */
  }
  x = 0.0;                                          /* initialize sum */
  for (i=0;i<q->numcl;i++) {                        /* for all classifications */
    ybar[i] = ymin/ybar[i];                         /* normalize values */
    x =+ ybar[i];                                   /* and sum them */
  }
  x = 1.0/x;                                        /* take the reciprocal */
  for (i=0;i<q->numcl;i++) {                        /* for all classifications */
    ybar[i] =* x;                                   /* set likelihood for type */
    ptr[i]->cf = ((1.0/(q->numvset))*(1.0/q->numcl)) +
      ((1.0-(1.0/(q->numvset))) * ybar[i]);/* calculate certainty fact */
  }
  return;                                           /* return to point of call */
}

/* This routine creates a class hypothesis and returns a pointer to the
   newly created node */
struct classh *crclhyp () {
  struct classh *p;
  p = (struct classh *) calloc (1,(sizeof(struct classh)));
    /* allocate storage for the new class hypothesis node */
  p->chigher = (struct fhypl *) calloc (1,(sizeof(struct fhypl)));
    /* allocate storage for the class hypoth list header node */
  p->fhtail = (struct fhypl *) calloc (1,(sizeof(struct fhypl)));
    /* allocate storage for the class hypoth list sentinel node */
```

```
  (p->chigher)->flnext = p->fhtail;                    /* frm fil class hyp lst */
  (p->chigher)->flprev = NIL;
  (p->fhtail)->flnext = NIL;
  (p->fhtail)->flprev = p->chigher;
  p->clower = (struct eventl *) calloc (1,(sizeof(struct eventl)));
    /* allocate storage for the event list header node */
  p->chtl = (struct eventl *) calloc (1,(sizeof(struct eventl)));
    /* allocate storage for the event list sentinel node */
  (p->clower)->elnext = p->chtl;                    /* form event list */
  (p->clower)->elprev = NIL;
  (p->chtl)->elnext = NIL;
  (p->chtl)->elprev = p->clower;
  return (p);                                      /* return pointer */
}

/* This routine removes a classification hypothesis node from the
   classification hypothesis list given a pointer to the node */
remchyp (p)
  struct classh *p; {
  struct eventl *q, *t1;
  struct fhypl *r, *t2;
  struct chypl *s;
  q = (p->clower)->elnext;                         /* point to 1st event ls node */
  while (q != p->chtl) {                           /* while not thru list, do */
    if (q->elink != NIL) {                         /* if not a NIL link */
      s = ((q->elink)->hgher)->cnxt;               /* pnt to 1st cl hyp list nod */
      while (s->hlink != p)                        /* while not to own node */
        s = s->cnxt;                               /* advance pointer */
      dlete14 (s);                                 /* remove node from list */
      cfree (s);                                   /* and free storage */
    }
    t1 = q;                                        /* set temporary pointer */
    q = q->elnext;                                 /* advance list pointer */
    dlete16 (t1);                                  /* delete node from the list */
    cfree (t1);                                    /* and free storage */
  }
```

```
  cfree (p->clower);                    /* remove header and */
  cfree (p->chtl);                      /* sentinel nodes too */
  r = (p->chigher)->flnext;             /* pnt to 1st fil hyp is node */
  while (r != p->fhtail) {              /* while not thru list, do */
    t2 = r;                             /* set temporary pointer */
    r = r->flnext;                      /* advance list pointer */
    dlete17 (t2);                       /* delete node from list */
    cfree (t2);                         /* and free storage */
  }
  cfree (p->chigher);                   /* remove header and */
  cfree (p->fhtail);                    /* sentinel nodes too */
  dlete15 (p);                          /* delete class node itself */
  cfree (p);                            /* and free the storage */
  return;                               /* return to point of call */
}
```

```
/* This module allows the user to enter information into the program */

#include <stdio.h>
#include "pgmvar.c"          /* include program data */
#include "exincld.c"         /* include external variables */

/* This routine is the control routine for the routines that allow
   the user to input information to the global data base - i.e. allows
   the user to input outside information or hypotheses about the
   situation, to delete information or hypotheses that are no longer
   current, to change the sensor configuration or change sensor
   parameters, to ask a question about a deduction the system made,
   or to stop the program. */
chkinpt () {
  int goarnd, rtnval;
  char answer;
  goarnd = TRUE;                          /* set loop condition to no stop */
  while (goarnd) {                        /* while not stop, do */
    printf("options:  a(dd hypothesis  r(emove hypothesis0); /* print */
    printf("          i(nsert sensor   d(elete sensor0);     /* user */
    printf("          c(hange sensor   p(rint a list0); /* instructions */
    printf("          w(hy ?           s(top program0);
    printf("          q(uit0);
    printf("choose one...");
    answer = getans ();                   /* get user response */
    switch (answer) {                     /* case stmt for answer */
      case 'a':
        addhyp ();                        /* add a hypothesis */
        break;
      case 'r':
        remhyp ();                        /* remove a hypothesis */
        break;
      case 'i':
        insrtsn ();                      /* insert a sensor */
        break;
      case 'd':
```

```
            deletsn ();                          /* delete a sensor */
            break;
          case 'c':
            chgsenp ();                          /* change a sensor */
            break;
          case 'p':
            prntopt ();                          /* print a list */
            break;
          case 'w':
            intract ();                          /* answer user questions */
            break;
          case 's':
            rtnval = STOP;                       /* set program stop condition */
            goarnd = FALSE;                      /* set stop loop condition */
            break;
          case 'g':
            rtnval = TRUE;                       /* continue program */
            goarnd = FALSE;                      /* set stop loop condition */
            break;
          default:
            printf("command not understood0); /* print error message */
            break;
        }
      }
      return (rtnval);                           /* return value to point of call */
}

/* This routine allows the user to add an outside information
   hypothesis */
addhyp () {
  printf("option not implemented0);
  return;                                        /* return to point of call */
}

/* This routine allows the user to remove an outside information
   hypothesis */
```

```
remhyp () {
  printf("option not implemented0);
  return;                                   /* return to point of call */
}

/* This routine allows the user to ask questions about system deductions */
intract () {
  printf("option not implemented0);
  return;                                   /* return to point of call */
}

/* This routine allows the user to insert a sensor */
insrtsn () {
  int sensnbr, i, araynbr, rtnval, nstp, goon, nogo;
  char ans;
  struct arayid *pointr, *chkalst ();
  struct sensnod *new, chkslst ();
  nogo = FALSE;                             /* set stop condition to no stop */
  nstp = TRUE;                              /* set loop condition to no stop */
  while (nstp) {                            /* while not stop, do */
    printf("does array in which sensor will be located0); /* print user */
    printf("already exist ? (y or n)");                   /* instructions */
    ans = getans ();                        /* get user response */
    if (ans == 'n') {                       /* if no */
      araynbr = makarry ();                 /* create a new array if possible */
      switch (araynbr) {                    /* case stmt for result */
        case TRMINAT:
          nstp = FALSE;                     /* set loop stopping condition */
          nogo = TRUE;                      /* terminate the insert try */
          printf("termination accepted0); /* inform user */
          break;
        case NIL:
          printf("do you still want to insert ? (y or n)"); /* user msg */
          ans = getans ();                  /* get user response */
          if (ans == 'n') {                 /* if no */
            nstp = FALSE;                   /* set loop stopping condition */
```

```
        nogo = TRUE;                    /* terminate the insert try */
        printf("termination accepted0); /* inform user */
      }
      else                              /* otherwise */
        printf("try again...0);    /* inform user of new try */
      break;
    default:
      nstp = FALSE;                     /* set loop stopping condition */
      printf("new array ID = %d0,araynbr); /* tell user new ID */
      pointr = chkalst (araynbr);   /* get pointer to new array */
      break;
  }
}
else {                                  /* otherwise */
  goon = TRUE;                          /* set loop condition to no stop */
  while (goon) {                        /* while not stop, do */
    araynbr = getidnm ();               /* get ID number from user */
    if ((pointr = chkalst (araynbr)) == NIL) { /* if bad ID */
      printf("no array associated with0); /* print error msg */
      printf("ID number given...0);
      printf("do you still want to insert ? (y or n)"); /* user instr */
      ans = getans ();                  /* get user response */
      if (ans == 'n') {                 /* if no */
        goon = FALSE;                   /* set loop stopping condition */
        nstp = FALSE;                   /* set loop stopping condition */
        nogo = TRUE;                    /* terminate insertion try */
      }
      else                              /* otherwise */
        goon = FALSE;                   /* set stopping condition */
    }
    else {                              /* otherwise, good ID number */
      goon = FALSE;                     /* set loop stopping condition */
      nstp = FALSE;                     /* set loop stopping condition */
    }
  }
}
```

```
  }
  if (nogo == FALSE) {                          /* if no termination */
    new = (struct sensnod *) calloc (1,(sizeof(struct sensnod)));
      /* allocate storage for new sensor node */
    rtnval = instsnd (new, pointr);             /* insert into sensor list in */
      /* proper position, link to array, increment number of sensors in */
      /* array (or return error) */
    if (rtnval == ERROR) {                       /* if error */
      printf("cannot create a sensor for that array0); /* print error */
      printf("limit already reached...0);               /* message */
      printf("insertion try terminated0); /* inform user */
      cfree (new);                               /* deallocate storage */
    }
    else {                                       /* otherwise */
      printf("new sensor being created0); /* inform user */
      sensnbr = getidnm ();                      /* get ID number from user */
      new->tmactiv = systime;                   /* set activation time */
      for (i=0;i<NWINDW;i++)                     /* set default conditions */
        new->numcnt[i] = 0;
      new->soil = DEFSOIL;
      new->hyprsen = systime;
      new->offon = systime;
      new->numcont = 0;
      new->coloc = NIL;
      new->expect = 1.0;
      intusdl (new);                            /* initialize unfiltered data ls */
      infsdll (new);                            /* initialize filtered data ls */
      initvsl (new);                             /* initialize valid set ls */
      sensnbr = gtsnval (new, sensnbr); /* get sensor characteristics */
        /* from user, insert into ID list */
      if ((new->parent)->numsen > 1)             /* check for adjacency */
        sadjcnt (new, pointr);                   /* if so, get from user */
    }
  }
  return;                                        /* return to point of call */
}
```

```
/* This routine is a temporary routine used to test program logic in
   dealing with unrecognized sensor types */
tempfix (new, val)
  struct sensnod *new;
  int val; {
  int i;
  new->type = val * 10;                          /* set type */
  new->accangl = 360.0;                          /* set acceptance angle */
  new->hs = 10.0;                                /* set inhibit time */
  new->eol = new->tmactiv + 864000.0;            /* EOL at 10 days */
  new->coloc = NIL;                              /* no colocated sensor */
  for (i=0;i<NTTYP;i++)
    new->detrad[i] = 50.0;                       /* set detection radius */
  return;                                        /* return to point of call */
}

/* This routine allows the user to delete a sensor node.  When a node
   is deleted, it is checked to see if its valid set list is empty.
   If the valid set list is empty, the node is deleted and the storage is
   deallocated.  If the list is not empty, then the node is moved to
   the deleted list and retained until its valid set list is empty. */
deletsn () {
  printf("option not implemented0);
  return;                                        /* return to point of call */
}

/* This routine allows the user to change sensor characteristics */
/* note that many of the subroutines to be used here were previously
   developed for the insert routine */
chgsenp () {
  printf("option not implemented0);
  return;
}

/* This routine allows the user to copy lists of nodes and sensor data
```

```
   to a scratch file, which is automatically sent to the line printer */
prntopt () {
  int nstp, val;
  char ans;
  nstp = TRUE;                              /* set loop condition to no stop */
  while (nstp) {                            /* while not stop, do */
    printf("options:  0)quit0);           /* user */
    printf("          1)sensor ID and node list0); /* instructions */
    printf("          2)array ID and node list0);
    printf("          3)field ID and node list0);
    printf("          4)root node0);
    printf("          5)sensor activations0);
    printf("          6)unfiltered data lists0);
    printf("          7)filtered data lists0);
    printf("          8)valid set lists0);
    printf("          9)event hypothesis node list0);
    printf("         10)classification hypothesis node list0);
    printf("         11)filtered class hypoth node list0);
    printf("         12)valid set list node lists0);
    printf("         13)event list node list0);
    printf("         14)event hypoth list node lists0);
    printf("0ress <return> to continue listing0);
    ans = getans ();                  /* any char continues listing */
    printf("         15)class hypoth list node lists0);
    printf("         16)filtered class hypoth node lists0);
    printf("         17)velocity list node lists0);
    printf("         18)deleted sensor ID list0);
    printf("         19)deleted array ID list0);
    printf("         20)deleted field ID list0);
    printf("         21)filtered data lists lists0);
    printf("enter the number of the option you wish"); /* ask user */
    val = getint ();                        /* get user response */
    switch (val) {                          /* case stmt for answer */
      case 0:
        nstp = FALSE;                       /* set stopping condition */
        break;
```

```
case 1:
  sdlhdcp ();                          /* sensor node and ID list */
  break;
case 2:
  adlhdcp ();                          /* array node and ID list */
  break;
case 3:
  fdlhdcp ();                          /* field node and ID list */
  break;
case 4:
  rthdcpy ();                           /* root node */
  break;
case 5:
  if (systime == 0.0)
    printf("no data generated yet0); /* data list */
  else
    dthdcpy ();
  break;
case 6:
  sufdlhc ();                           /* unfiltered data lists */
  break;
case 7:
  sfdlhc ();                            /* filtered data lists */
  break;
case 8:
  vslhc ();                             /* valid set lists */
  break;
case 9:
  ehnhc ();                             /* event hypoth node list */
  break;
case 10:
  chnhc ();                             /* class hypoth node list */
  break;
case 11:
  fchnhc ();                           /* filterd class hyp node list */
  break;
```

```
      case 12:
        vslnlhc ();                          /* valid set list node list */
        break;
      case 13:
        elnlhc ();                           /* event list node list */
        break;
      case 14:
        ehlnlhc ();                          /* evnt hyp list node lists */
        break;
      case 15:
        chlnlhc ();                          /* class hyp list node lists */
        break;
      case 16:
        fchlnlc ();                          /* fltrd class hyp lst node list */
        break;
      case 17:
        vlnlhc ();                           /* velocity list node lists */
        break;
      case 18:
        dsnlhc ();                           /* deleted sensor ID list */
        break;
      case 19:
        darlhc ();                           /* deleted array ID list */
        break;
      case 20:
        dfllhc ();                           /* deleted field ID list */
        break;
      case 21:
        sfdllhc ();                          /* filtered data lists lists */
        break;
      default:
        printf("command not understood0); /* print error message */
        break;
    }
  }
  return;                                    /* return to point of call */
```

```
/* This module contains the routines used to print all manner of lists
   in hard copy form */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "genvar.c"                          /* include generator data */
#include "exincld.c"                         /* include external variables */

/* This routine sends a copy of the sensor ID list and the sensor node
   list to a scratch file and then calls the system automatically to
   print the file */
sdlhdcp () {
  int i;
  struct sensid *p;
  struct sensnod *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)   /* open file if possible */
    printf("unable to open scratch file0); /* print error message */
  else {                                   /* otherwise */
    fprintf(fp,"at time = %f0,systime); /* write system time */
    fprintf(fp,"sensor ID and node list0); /* header info */
    p = sidhdr->nxt;                       /* point to first ID node */
    while (p != sidtl) {                   /* while not through, do */
      q = p->indic;                        /* point to sensor node */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0dno = %d0ndic = %d0xt = %d0,
        p,p->idno,p->indic,p->nxt);
      fprintf(fp,"associated sensor node0);
      fprintf(fp,"address = %d0bmapno = %d type = %d0,q,q->sbmapno,
        q->type);
      fprintf(fp,"slocx = %f slocy = %f detdir = %f0,
        q->slocx,q->slocy,q->detdir);
      fprintf(fp,"detrad =0);
      for (i=0;i<(NTTYP/2);i++)
        fprintf(fp,"%f    %f0,q->detrad[2*i],q->detrad[2*i+1]);
      fprintf(fp,"accangl = %f tmactiv = %f hs = %f0,
        q->accangl,q->tmactiv,q->hs);
```

```
      fprintf(fp,"expect = %f0,q->expect);
      fprintf(fp,"numcnt =0);
      for (i=0;i<(NWINDW/2);i++)
        fprintf(fp,"%d    %d0,q->numcnt[2*i],q->numcnt[2*i+1]);
      fprintf(fp,"soil = %d hyprsen = %f offon = %f0,
        q->soil,q->hyprsen,q->offon);
      fprintf(fp,"numcont = %d eol = %f0,q->numcont,q->eol);
      fprintf(fp,"coloc = %d head = %d tail = %d0,q->coloc,q->head,
        q->tail);
      fprintf(fp,"hd = %d tl = %d0,q->hd,q->tl);
      fprintf(fp,"hdr = %d sentl = %d0,q->hdr,q->sentl);
      fprintf(fp,"parent = %d prev = %d0,q->parent,q->prev);
      fprintf(fp,"nx = %d sidback = %d0,q->nx,q->sidback);
      p = p->nxt;                          /* advance pointer */
    }
    fclose (fp);                           /* close the file */
    system ("list HARDCOPY |lpr");         /* call system to print list */
    printf("sensor ID and node list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                  /* return to point of call */
}

/* This routine sends a copy of the array ID list and the array node
   list to a scratch file and then calls the system automatically to
   print the file */
adlhdcp () { /* array node and id list */
return;
}

/* This routine sends a copy of the field ID list and the field node
   list to a scratch file and then calls the system automatically to
   print the file */
fdlhdcp () { /* field node and id list */
return;
}
```

```
/* This routine sends a copy of the root node to a scratch file and then
   calls the system automatically to print the file */
rthdcpy () { /* root node */
return;
}

/* This routine sends a copy of the data list generated by the generator
   routine to a scratch file and then calls the system automatically to
   print the file */
dthdcpy () { /* data list */
return;
}

/* This routine sends a copy of the sensor unfiltered data list to a
   scratch file and then calls the system automatically to print
   the file */
sufdlhc () { /* unfiltered data lists */
return;
}

/* This routine sends a copy of the sensor filtered data lists to a
   scratch file and then calls the system automatically to print
   the file */
sfdlhc () { /* filtered data lists */
return;
}

/* This routine sends a copy of the sensor valid set list to a
   scratch file and then calls the system automatically to print
   the file */
vslhc () { /* valid set lists */
return;
}

/* This routine sends a copy of the event hypothesis node list to a
```

```
   scratch file and then calls the system automatically to print
   the file */
ehnhc () {
  struct evnt *p;
  if ((fp = fopen(pfilnam,"w")) == NULL) /* open scratch file */
    printf("unable to open scratch file0);
  else {                                 /* if opened */
    p = evhdr->fll;                      /* point to 1st node in list */
    fprintf(fp,"at time = %f0,systime);  /* write time to file */
    fprintf(fp,"event hypothesis node list0); /* header info */
    while (p != evtl) {                  /* while not thru list, do */
      fprintf(fp,"event hypothesis node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"etyp = %d0,p->etyp);
      fprintf(fp,"velmax = %f velmin = %f velav = %f0,
        p->velmax,p->velmin,p->velav);
      fprintf(fp,"numvset = %d numcl = %d updtim = %f0,
        p->numvset,p->numcl,p->updtim);
      fprintf(fp,"last = %d fll = %d0,p->last,p->fll);
      fprintf(fp,"lowr = %d vstail = %d0,p->lowr,p->vstail);
      fprintf(fp,"hgher = %d chtail = %d0,p->hgher,p->chtail);
      fprintf(fp,"vhdr = %d vsent = %d0,p->vhdr,p->vsent);
      p = p->fll;                        /* advance pointer */
    }
    fclose (fp);                         /* close the scratch file */
    system ("list HARDCOPY |lpr");       /* print the file */
    printf("event hypothesis node list sent to the printer0); /* user */
    printf("with the name of HARDCOPY0);
  }
  return;                                /* return to point of call */
}

/* This routine sends a copy of the classification hypothesis node list to
   a scratch file and then calls the system automatically to print
   the file */
chnhc () {
```

```
  struct classh *p;
  if ((fp = fopen(pfilnam,"w")) == NULL) /* open scratch file */
    printf("unable to open scratch file0);
  else {                                  /* if opened */
    p = clhdr->hnext;                      /* point to 1st node in list */
    fprintf(fp,"at time = %f0,systime);   /* write time to file */
    fprintf(fp,"classification hypothesis node list0); /* header info */
    while (p != cltl) {                    /* while not thru list, do */
      fprintf(fp,"classification hypothesis node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"ctyp = %d0,p->ctyp);
      fprintf(fp,"class = %d cvel = %f cdir = %f0,
        p->class,p->cvel,p->cdir);
      fprintf(fp,"clength = %f cnum = %d cloc = %d0,
        p->clength,p->cnum,p->cloc);
      fprintf(fp,"ceftim = %f cupd = %f cf = %f0,
        p->ceftim,p->cupd,p->cf);
      fprintf(fp,"hprev = %d hnext = %d0,p->hprev,p->hnext);
      fprintf(fp,"clower = %d chtl = %d0,p->clower,p->chtl);
      fprintf(fp,"chigher = %d fhtail = %d0,p->chigher,p->fhtail);
      p = p->hnext;                        /* advance pointer */
    }
    fclose (fp);                           /* close the scratch file */
    system ("list HARDCOPY |lpr");         /* print the file */
    printf("classification hypothesis node list sent to the printer0);
    printf("with the name of HARDCOPY0);  /* user msg */
  }
  return;                                 /* return to point of call */
}

/* This routine sends a copy of the filtered class hypothesis node list
   to a scratch file and then calls the system automatically to print
   the file */
fchnhc () { /* filtered class hyp node list */
return;
}
```

```
/* This routine sends a copy of the deleted sensor ID list to a
   scratch file and then calls the system automatically to print
   the file */
dsnlhc () { /* deleted sensor id list */
return;
}

/* This routine sends a copy of the deleted array ID list to a
   scratch file and then calls the system automatically to print
   the file */
darlhc () { /* deleted array id list */
return;
}

/* This routine sends a copy of the deleted field ID list to a
   scratch file and then calls the system automatically to print
   the file */
dfllhc () { /* deleted field id list */
return;
}

/* This routine sends a copy of the event hypothesis list node list to
   a scratch file and then calls the system automatically to print
   the file */
ehlnlhc () { /* evnt hyp list node lists */
return;
}

/* This routine sends a copy of the valid set list nodes list to a
   scratch file and then calls the system automatically to print
   the file */
vslnlhc () { /* valid set list node list */
return;
}
```

```
/* This routine sends a copy of the event list nodes list to a
   scratch file and then calls the system automatically to print
   the file */
elnlhc () { /* event list node list */
return;
}

/* This routine sends a copy of the filtered classification hypothesis
   list nodes list to a scratch file and then calls the system
   automatically to print the file */
fchlnlc () { /* fltrd class hyp lst node list */
return;
}

/* This routine sends a copy of the classification hypothesis list nodes
   list to a scratch file and then calls the system automatically to print
   the file */
chlnlhc () { /* class hyp list node lists */
return;
}

/* This routine sends a copy of the velocity list nodes list to a
   scratch file and then calls the system automatically to print
   the file */
vlnlhc () { /* velocity list node lists */
return;
}

/* This routine sends a copy of the sensor filtered data lists list to a
   scratch file and then calls the system automatically to print
   the file */
sfdllhc () { /* filtered data lists lists */
return;
}
```

```
/* This module contains the routines used to print all manner of lists
   in hard copy form.  This module is the storehouse for all
   list-to-printer routines as only certain print routines are
   contained in the execution module to conserve space.  When a print
   routine needs to be added to the execution module, a copy is made
   from this module and inserted in listprt.c */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "genvar.c"                          /* include generator data */
#include "exincld.c"                         /* include external variables */

/* This routine sends a copy of the sensor ID list and the sensor node
   list to a scratch file and then calls the system automatically to
   print the file */
sdlhdcp () {
  int i;
  struct sensid *p;
  struct sensnod *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)   /* open file if possible */
    printf("unable to open scratch file0); /* print error message */
  else {                                   /* otherwise */
    fprintf(fp,"at time = %f0,systime); /* write system time */
    fprintf(fp,"sensor ID and node list0); /* header info */
    p = sidhdr->nxt;                       /* point to first ID node */
    while (p != sidtl) {                   /* while not through, do */
      q = p->indic;                        /* point to sensor node */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0dno = %d0ndic = %d0xt = %d0,
        p,p->idno,p->indic,p->nxt);
      fprintf(fp,"associated sensor node0);
      fprintf(fp,"address = %d0bmapno = %d type = %d0,q,q->sbmapno,
        q->type);
      fprintf(fp,"slocx = %f slocy = %f detdir = %f0,
        q->slocx,q->slocy,q->detdir);
      fprintf(fp,"detrad =0);
```

```
        for (i=0;i<(NTTYP/2);i++)
          fprintf(fp,"%f    %f0,q->detrad[2*i],q->detrad[2*i+1]);
        fprintf(fp,"accangl = %f tmactiv = %f hs = %f0,
          q->accangl,q->tmactiv,q->hs);
        fprintf(fp,"expect = %f0,q->expect);
        fprintf(fp,"numcnt =0);
        for (i=0;i<(NWINDW/2);i++)
          fprintf(fp,"%d    %d0,q->numcnt[2*i],q->numcnt[2*i+1]);
        fprintf(fp,"soil = %d hyprsen = %f offon = %f0,
          q->soil,q->hyprsen,q->offon);
        fprintf(fp,"numcont = %d eol = %f0,q->numcont,q->eol);
        fprintf(fp,"coloc = %d head = %d tail = %d0,q->coloc,q->head,
          q->tail);
        fprintf(fp,"hd = %d tl = %d0,q->hd,q->tl);
        fprintf(fp,"hdr = %d sentl = %d0,q->hdr,q->sentl);
        fprintf(fp,"parent = %d prev = %d0,q->parent,q->prev);
        fprintf(fp,"nx = %d sidback = %d0,q->nx,q->sidback);
        p = p->nxt;                          /* advance pointer */
      }
      fclose (fp);                           /* close the file */
      system ("list HARDCOPY |lpr");         /* call system to print list */
      printf("sensor ID and node list sent to the printer0); /* user msg */
      printf("with the name of HARDCOPY0);
    }
    return;                                  /* return to point of call */
}

/* This routine sends a copy of the array ID list and the array node
   list to a scratch file and then calls the system automatically to
   print the file */
adlhdcp () {
  int i;
  struct arayid *p;
  struct araynod *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)       /* open file if possible */
    printf("unable to open scratch file0); /* print error message */
```

```
  else {                                          /* otherwise */
    fprintf(fp,"at time = %f0,systime);    /* write system time */
    fprintf(fp,"array ID and node list0);/* header info */
    p = aidhdr->next;                             /* point to first ID node */
    while (p != aidtl) {                          /* while not through, do */
      q = p->pntr;                                /* point to array node */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0raynum = %d0ntr = %d0ext = %d0,
        p,p->araynum,p->pntr,p->next);
      fprintf(fp,"associated array node0);
      fprintf(fp,"address = %d0bmapno = %d typ = %d numsen = %d0,
        q,q->abmapno,q->typ,q->numsen);
      fprintf(fp,"vmin =0);
      for (i=0;i<(NTTYP/2);i++)
        fprintf(fp,"%f    %f0,q->vmin[2*i],q->vmin[2*i+1]);
      fprintf(fp,"vmax =0);
      for (i=0;i<(NTTYP/2);i++)
        fprintf(fp,"%f    %f0,q->vmax[2*i],q->vmax[2*i+1]);
      fprintf(fp,"vav =0);
      for (i=0;i<(NTTYP/2);i++)
        fprintf(fp,"%f    %f0,q->vav[2*i],q->vav[2*i+1]);
      fprintf(fp,"sadajc =0);
      for (i=0;i<(NBYTES/2);i++)
        fprintf(fp,"%o    %o0,q->sadajc[2*i],q->sadajc[2*i+1]);
      fprintf(fp,"headr = %d sntinl = %d0,q->headr,q->sntinl);
      fprintf(fp,"lf = %d rt = %d0,q->lf,q->rt);
      fprintf(fp,"parnt = %d aidback = %d0,q->parnt,q->aidback);
      p = p->next;                              /* advance pointer */
    }
    fclose (fp);                                  /* close the file */
    system ("list HARDCOPY |lpr");                /* call system to print list */
    printf("array ID and node list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                         /* return to point of call */
}
```

```
/* This routine sends a copy of the field ID list and the field node
   list to a scratch file and then calls the system automatically to
   print the file */
fdlhdcp () {
  int i;
  struct fildid *p;
  struct fildnod *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)        /* open file if possible */
    printf("unable to open scratch file0); /* print error message */
  else {                                          /* otherwise */
    fprintf(fp,"at time = %f0,systime);     /* write system time */
    fprintf(fp,"field ID and node list0);/* header info */
    p = flidhdr->follow;                          /* point to first ID node */
    while (p != flidtl) {                         /* while not through, do */
      q = p->ptr;                                 /* point to field node */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0ildnum = %d0tr = %d0ollow = %d0,
        p,p->fildnum,p->ptr,p->follow);
      fprintf(fp,"associated field node0);
      fprintf(fp,"address = %d0eldtyp = %d fbmapno = %d numary = %d0,
        q,q->feldtyp,q->fbmapno,q->numary);
      fprintf(fp,"aradajc =0);
      for (i=0;i<(NBYTES/2);i++)
        fprintf(fp,"%o    %o0,q->aradajc[2*i],q->aradajc[2*i+1]);
      fprintf(fp,"header = %d sntnl = %d0,q->header,q->sntnl);
      fprintf(fp,"fprev = %d fnext = %d0,q->fprev,q->fnext);
      fprintf(fp,"rootpar = %d fidback = %d0,q->rootpar,q->fidback);
      p = p->follow;                            /* advance pointer */
    }
    fclose (fp);                                /* close the file */
    system ("list HARDCOPY !lpr");              /* call system to print list */
    printf("field ID and node list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                       /* return to point of call */
```

```
}

/* This routine sends a copy of the root node to a scratch file and then
   calls the system automatically to print the file */
rthdcpy () {
  int i;
  struct fldroot *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)        /* open file if possible */
    printf("unable to open scratch file0); /* print error message */
  else {                                          /* otherwise */
    fprintf(fp,"at time = %f0,systime);     /* write system time */
    fprintf(fp,"root node0);                /* header info */
      q = rootptr;                                /* point to root node */
      fprintf(fp,"address = %d0umfld = %d0,q,q->numfld);
      fprintf(fp,"fadajc =0);
      for (i=0;i<(NBYTES/2);i++)
        fprintf(fp,"%o    %o0,q->fadajc[2*i],q->fadajc[2*i+1]);
      fprintf(fp,"fldhdr = %d fldtl = %d0,q->fldhdr,q->fldtl);
    fclose (fp);                                  /* close the file */
    system ("list HARDCOPY |lpr");                /* call system to print list */
    printf("root node sent to the printer0);      /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                         /* return to point of call */
}

/* This routine sends a copy of the data list generated by the generator
   routine to a scratch file and then calls the system automatically to
   print the file */
dthdcpy () {
  char bdata;
  int i, j, k;
  float tym;
  FILE *fpo;
  if ((fp = fopen(dfilnam,"r")) == NULL)     /* attempt to open data file */
    printf("unable to open data file0);
```

```
  else                                          /* if opened */
    if ((fpo = fopen(pfilnam,"w")) == NULL)     /* open scratch file */
      printf("unable to open scratch file0);
    else {                                       /* if opened */
      i = (TMWINDW * NPERSEC) / 6;              /* calculate bytes to read */
      i = (((TMWINDW * NPERSEC) % 6) == 0 ? i : i + 1);
      fscanf(fp,"%f",&tym);                      /* read generation time */
      fprintf(fpo,"at time = %f0eneration base time = %f0,
        systime,tym);
      fprintf(fpo,"sensor data list0);          /* header info */
      for (j=0;j<i;j++) {                        /* for all data bytes */
        for (k=0;k<senscnt;k++) {                /* for all sensors */
          fscanf(fp,"%c",&bdata);                /* read byte of data */
          fprintf(fpo,"%o ",bdata);              /* write to file */
        }
        fprintf(fpo,"0);                        /* new line between loops */
      }
      fclose (fp);                               /* close the data file */
      fclose (fpo);                              /* close the scratch file */
      system ("list HARDCOPY !lpr");             /* send to printer */
      printf("sensor activations sent to the printer0); /* user msg */
      printf("with the name of HARDCOPY0);
    }
  return;                                        /* return to point of call */
}

/* This routine sends a copy of the sensor unfiltered data list to a
   scratch file and then calls the system automatically to print
   the file */
sufdlhc () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatau *r;
  if ((fp = fopen(pfilnam,"w")) == NULL)          /* open scratch file */
    printf("unable to open scratch file0);
  else {                                         /* if opened */
```

```
    p = sidhdr->nxt;                                  /* set pntr to 1st node */
    fprintf(fp,"at time = %f0,systime);     /* write time to file */
    fprintf(fp,"unfiltered data list0);       /* header info */
    while (p != sidtl) {                              /* while nodes left, do */
      q = p->indic;                                   /* point to sensor node */
      r = (q->head)->right;                           /* pnt to 1st uf data nd */
      fprintf(fp,"for sensor %d0,p->idno);
      while (r != q->tail) {                          /* for all uf data nodes */
        fprintf(fp,"unfiltered data node0);
        fprintf(fp,"address = %d0,r);
        fprintf(fp,"timact = %f0,r->timact);
        fprintf(fp,"left = %d right = %d0,r->left,r->right);
        fprintf(fp,"prnt = %d0,r->prnt);
        r = r->right;                                 /* advance pointer */
      }
      p = p->nxt;                                     /* advance pointer */
    }
    fclose (fp);                                      /* close the scratch file */
    system ("list HARDCOPY |lpr");                    /* print the file */
    printf("unfiltered data lists sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                             /* return to point of call */
}

/* This routine sends a copy of the sensor filtered data lists to a
   scratch file and then calls the system automatically to print
   the file */
sfdlhc () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatafl *r;
  struct sdataf *s;
  if ((fp = fopen(pfilnam,"w")) == NULL)          /* open scratch file */
    printf("unable to open scratch file0);
  else {                                          /* if opened */
```

```
    p = sidhdr->nxt;                                    /* set pntr to 1st node */
    fprintf(fp,"at time = %f0,systime);     /* write time to file */
    fprintf(fp,"filtered data lists0);       /* header info */
    while (p != sidtl) {                                /* while nodes left, do */
      q = p->indic;                                     /* point to sensor node */
      r = (q->hd)->fdlnx;                              /* pnt to 1st fil data ls nd */
      fprintf(fp,"for sensor %d0,p->idno);
      while (r != q->tl) {                             /* while not thru list, do */
        s = (r->dfhead)->rght;                         /* point to 1st node */
        fprintf(fp,"for type filtering = %d0,
          r->fdltyp);
        while (s != r->dfsent) {                        /* for all fil data nodes */
          fprintf(fp,"filtered data node0);
          fprintf(fp,"address = %d0,s);
          fprintf(fp,"tmact = %f0,s->tmact);
          fprintf(fp,"lft = %d rght = %d0,s->lft,s->rght);
          fprintf(fp,"pnt = %d0,s->pnt);
          s = s->rght;                                   /* advance pointer */
        }
        r = r->fdlnx;                                    /* advance pointer */
      }
      p = p->nxt;                                       /* advance pointer */
    }
    fclose (fp);                                        /* close the scratch file */
    system ("list HARDCOPY |lpr");                      /* print the file */
    printf("filtered data lists sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                               /* return to point of call */
}

/* This routine sends a copy of the sensor valid set list to a
   scratch file and then calls the system automatically to print
   the file */
vslhc () {
  struct sensid *p;
```

```
  struct sensnod *q;
  struct vset *r;
  if ((fp = fopen(pfilnam,"w")) == NULL)          /* open scratch file */
    printf("unable to open scratch file0);
  else {                                          /* if opened */
    p = sidhdr->nxt;                              /* set pntr to 1st node */
    fprintf(fp,"at time = %f0,systime);     /* write time to file */
    fprintf(fp,"valid set list0);       /* header info */
    while (p != sidtl) {                          /* while nodes left, do */
      q = p->indic;                               /* point to sensor node */
      r = (q->hdr)->foll;                        /* pnt to 1st valid set nd */
      fprintf(fp,"for sensor %d0,p->idno);
      while (r != q->sentl) {                     /* for all valid set nodes */
        fprintf(fp,"valid set list node0);
        fprintf(fp,"address = %d0,r);
        fprintf(fp,"numact = %d centrd = %f0,r->numact,r->centrd);
        fprintf(fp,"timupd = %f compl = %d0,r->timupd,r->compl);
        fprintf(fp,"lnk = %d lst = %d0,r->lnk,r->lst);
        fprintf(fp,"prevs = %d foll = %d0,r->prevs,r->foll);
        fprintf(fp,"higher = %d eltail = %d0,r->higher,r->eltail);
        fprintf(fp,"left = %d right = %d0,r->left,r->right);
        r = r->foll;                             /* advance pointer */
      }
      p = p->nxt;                                 /* advance pointer */
    }
    fclose (fp);                                  /* close the scratch file */
    system ("list HARDCOPY |lpr");                /* print the file */
    printf("valid set lists sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                         /* return to point of call */
}

/* This routine sends a copy of the event hypothesis node list to a
   scratch file and then calls the system automatically to print
   the file */
```

```
ehnhc () {
  struct evnt *p;
  if ((fp = fopen(pfilnam,"w")) == NULL) /* open scratch file */
    printf("unable to open scratch file0);
  else {                                  /* if opened */
    p = evhdr->fll;                       /* point to 1st node in list */
    fprintf(fp,"at time = %f0,systime);   /* write time to file */
    fprintf(fp,"event hypothesis node list0); /* header info */
    while (p != evtl) {                   /* while not thru list, do */
      fprintf(fp,"event hypothesis node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"etyp = %d0,p->etyp);
      fprintf(fp,"velmax = %f velmin = %f velav = %f0,
        p->velmax,p->velmin,p->velav);
      fprintf(fp,"numvset = %d numcl = %d updtim = %f0,
        p->numvset,p->numcl,p->updtim);
      fprintf(fp,"last = %d fll = %d0,p->last,p->fll);
      fprintf(fp,"lowr = %d vstail = %d0,p->lowr,p->vstail);
      fprintf(fp,"hgher = %d chtail = %d0,p->hgher,p->chtail);
      fprintf(fp,"vhdr = %d vsent = %d0,p->vhdr,p->vsent);
      p = p->fll;                         /* advance pointer */
    }
    fclose (fp);                          /* close the scratch file */
    system ("list HARDCOPY |lpr");        /* print the file */
    printf("event hypothesis node list sent to the printer0); /* user */
    printf("with the name of HARDCOPY0);
  }
  return;                                 /* return to point of call */
}

/* This routine sends a copy of the classification hypothesis node list to
   a scratch file and then calls the system automatically to print
   the file */
chnhc () {
  struct classh *p;
  if ((fp = fopen(pfilnam,"w")) == NULL) /* open scratch file */
```

```
    printf("unable to open scratch file0);
  else {                                        /* if opened */
    p = clhdr->hnext;                           /* point to 1st node in list */
    fprintf(fp,"at time = %f0,systime);         /* write time to file */
    fprintf(fp,"classification hypothesis node list0); /* header info */
    while (p != cltl) {                         /* while not thru list, do */
      fprintf(fp,"classification hypothesis node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"ctyp = %d0,p->ctyp);
      fprintf(fp,"class = %d cvel = %f cdir = %f0,
        p->class,p->cvel,p->cdir);
      fprintf(fp,"clength = %f cnum = %d cloc = %d0,
        p->clength,p->cnum,p->cloc);
      fprintf(fp,"ceftim = %f cupd = %f cf = %f0,
        p->ceftim,p->cupd,p->cf);
      fprintf(fp,"hprev = %d hnext = %d0,p->hprev,p->hnext);
      fprintf(fp,"clower = %d chtl = %d0,p->clower,p->chtl);
      fprintf(fp,"chigher = %d fhtail = %d0,p->chigher,p->fhtail);
      p = p->hnext;                             /* advance pointer */
    }
    fclose (fp);                                /* close the scratch file */
    system ("list HARDCOPY |lpr");              /* print the file */
    printf("classification hypothesis node list sent to the printer0);
    printf("with the name of HARDCOPY0);  /* user msg */
  }
  return;                                       /* return to point of call */
}

/* This routine sends a copy of the filtered class hypothesis node list
   to a scratch file and then calls the system automatically to print
   the file */
fchnhc () {
  struct filhyp *p;
  if ((fp = fopen(pfilnam,"w")) == NULL) /* open scratch file */
    printf("unable to open scratch file0);
  else {                                        /* if opened */
```

```
    p = filhdr->fright;                          /* point to 1st node in list */
    fprintf(fp,"at time = %f0,systime);   /* write time to file */
    fprintf(fp,"filtered classification hypothesis node list0);
    while (p != filtl) {                         /* while not thru list, do */
      fprintf(fp,"filtered classification hypothesis node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"fleft = %d fright = %d0,p->fleft,p->fright);
      p = p->fright;                               /* advance pointer */
    }
    fclose (fp);                                 /* close the scratch file */
    system ("list HARDCOPY !lpr");               /* print the file */
    printf("filtered classification hypothesis node list sent to the0);
    printf("printer with the name of HARDCOPY0);
  }
  return;                                        /* return to point of call */
}

/* This routine sends a copy of the deleted sensor ID list to a
   scratch file and then calls the system automatically to print
   the file */
dsnlhc () {
  struct dsensid *p;
  if ((fp = fopen(pfilnam,"w")) == NULL)     /* open scratch file */
    printf("unable to open scratch file0);
  else {                                         /* if opened */
    p = dsidhdr->dnxt;                           /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"deleted sensor ID list0);
    while (p != dsidtl) {                        /* while not at end, do */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"didno = %d dindic = %d dnxt = %d0,
        p->didno,p->dindic,p->dnxt);
      p = p->dnxt;                               /* advance pointer */
    }
    fclose (fp);                                 /* close the file */
```

```
    system ("list HARDCOPY |lpr");              /* print the file */
    printf("deleted sensor ID list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                       /* return to point of call */
}

/* This routine sends a copy of the deleted array ID list to a
   scratch file and then calls the system automatically to print
   the file */
darlhc () {
  struct darayid *p;
  if ((fp = fopen(pfilnam,"w")) == NULL)        /* open scratch file */
    printf("unable to open scratch file0);
  else {                                        /* if opened */
    p = daidhdr->dnext;                          /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"deleted array ID list0);
    while (p != daidtl) {                       /* while not at end, do */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"daraynum = %d dpntr = %d dnext = %d0,
        p->daraynum,p->dpntr,p->dnext);
      p = p->dnext;                              /* advance pointer */
    }
    fclose (fp);                                /* close the file */
    system ("list HARDCOPY |lpr");              /* print the file */
    printf("deleted array ID list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                       /* return to point of call */
}

/* This routine sends a copy of the deleted field ID list to a
   scratch file and then calls the system automatically to print
   the file */
```

```
dfllhc () {
  struct dfildid *p;
  if ((fp = fopen(pfilnam,"w")) == NULL)     /* open scratch file */
    printf("unable to open scratch file0);
  else {                                     /* if opened */
    p = dfidhdr->dfollow;                    /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"deleted field ID list0);
    while (p != dfidtl) {                    /* while not at end, do */
      fprintf(fp,"ID list node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"dfildnum = %d dptr = %d dfollow = %d0,
        p->dfildnum,p->dptr,p->dfollow);
      p = p->dfollow;                           /* advance pointer */
    }
    fclose (fp);                             /* close the file */
    system ("list HARDCOPY |lpr");           /* print the file */
    printf("deleted field ID list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                    /* return to point of call */
}

/*****************************************/
/* This routine sends a copy of the target track list to a
   scratch file and then calls the system automatically to print
   the file */
ttlhc () {
  struct tgttrk *p;
  if ((fp = fopen(pfilnam,"w")) == NULL)     /* open scratch file */
    printf("unable to open scratch file0);
  else {                                     /* if opened */
    p = thead->tnext;                       /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"target track list0);
    while (p != ttail) {                    /* while not at end, do */
```

```
      fprintf(fp,"target track node0);
      fprintf(fp,"address = %d0,p);
      fprintf(fp,"sensrno = %d nactv = %d0,p->sensrno,p->nactv);
      fprintf(fp,"lstact = %f lstchk = %f0,p->lstact,p->lstchk);
      fprintf(fp,"tnext = %d0,p->tnext);
      p = p->tnext;                          /* advance pointer */
    }
    fclose (fp);                             /* close the file */
    system ("list HARDCOPY |lpr");           /* print the file */
    printf("target track list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                    /* return to point of call */
}
/*****************************************/

/* This routine sends a copy of the event hypothesis list node list to
   a scratch file and then calls the system automatically to print
   the file */
ehlnlhc () {
  struct sensid *p;
  struct sensnod *q;
  struct vset *r;
  struct evntl *s;
  if ((fp = fopen(pfilnam,"w")) == NULL) /* open scratch file */
    printf("unable to open scratch file0);
  else {                                 /* if opened */
    p = sidhdr->nxt;                     /* set pointer to 1st list node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"event hypothesis list node list0);
    while (p != sidtl) {                 /* while not thru list, do */
      q = p->indic;                      /* point to sensor node */
      r = (q->hdr)->foll;                 /* point to 1st valid set */
      fprintf(fp,"for sensor %d0,p->idno);
      while (r != q->sentl) {            /* while not thru list, do */
        s = (r->higher)->follw;          /* point to 1st list node */
```

```
        fprintf(fp,"for valid set at %d0,r);
        while (s != r->eltail) {          /* while not thru list, do */
          fprintf(fp,"event hypothesis list node0);
          fprintf(fp,"address = %d0,s);
          fprintf(fp,"link = %d prvious = %d0,
            s->link,s->prvious);
          fprintf(fp,"follw = %d vlower = %d0,
            s->follw,s->prvious);
          s = s->follw;                   /* advance list node */
        }
        r = r->foll;                      /* advance list node */
      }
      p = p->nxt;                         /* advance list node */
    }
    fclose (fp);                          /* close the file */
    system ("list HARDCOPY |lpr");        /* print the file */
    printf("event hypothesis list node list sent to the printer0);
    printf("with the name of HARDCOPY0); /* user msg */
  }
  return;                                 /* return to point of call */
}

/* This routine sends a copy of the valid set list nodes list to a
   scratch file and then calls the system automatically to print
   the file */
vslnlhc () {
  struct evnt *p;
  struct vsetl *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)  /* open scratch file */
    printf("unable to open scratch file0);
  else {                                   /* if opened */
    p = evhdr->fll;                        /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"valid set list node list0);
    while (p != evtl) {                    /* while nodes left, do */
      q = (p->lowr)->lfoll;                 /* point to 1st node */
```

```
      fprintf(fp,"for event hypothesis at %d0,p);
      while (q != p->vstail) {              /* while not thru list, do */
        fprintf(fp,"valid set list node0);
        fprintf(fp,"address = %d0,q);
        fprintf(fp,"vslink = %d lprev = %d0,q->vslink,q->lprev);
        fprintf(fp,"lfoll = %d eparnt = %d0,
          q->lfoll,q->eparnt);
        q = q->lfoll;                       /* advance pointer */
      }
      p = p->fll;                           /* advance pointer */
    }
    fclose (fp);                            /* close the file */
    system ("list HARDCOPY |lpr");          /* print the file */
    printf("valid set list node list sent to the printer0);
    printf("with the name of HARDCOPY0); /* user msg */
  }
  return;                                   /* return to point of call */
}

/* This routine sends a copy of the event list nodes list to a
   scratch file and then calls the system automatically to print
   the file */
elnlhc () {
  struct classh *p;
  struct eventl *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)  /* open scratch file */
    printf("unable to open scratch file0);
  else {                                    /* if opened */
    p = clhdr->hnext;                         /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"event list node list0);
    while (p != cltl) {                     /* while nodes left, do */
      q = (p->clower)->elnext;                  /* point to 1st node */
      fprintf(fp,"for classification hypothesis at %d0,p);
      while (q != p->chtl) {              /* while not thru list, do */
        fprintf(fp,"event list node0);
```

```
        fprintf(fp,"address = %d0,q);
        fprintf(fp,"elink = %d elprev = %d0,q->elink,q->elprev);
        fprintf(fp,"elnext = %d chparnt = %d0,
          q->elnext,q->chparnt);
        q = q->elnext;                          /* advance pointer */
      }
      p = p->hnext;                             /* advance pointer */
    }
    fclose (fp);                                /* close the file */
    system ("list HARDCOPY |lpr");              /* print the file */
    printf("event list node list sent to the printer0);
    printf("with the name of HARDCOPY0); /* user msg */
  }
  return;                                       /* return to point of call */
}

/* This routine sends a copy of the filtered classification hypothesis
   list nodes list to a scratch file and then calls the system
   automatically to print the file */
fchlnlc () {
  struct classh *p;
  struct fhypl *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)  /* open scratch file */
    printf("unable to open scratch file0);
  else {                                        /* if opened */
    p = clhdr->hnext;                           /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"filtered classification hypothesis list node list0);
    while (p != cltl) {                         /* while nodes left, do */
      q = (p->chigher)->flnext;                 /* point to 1st node */
      fprintf(fp,"for classification hypothesis at %d0,p);
      while (q != p->fhtail) {                  /* while not thru list, do */
        fprintf(fp,"filtered classification hypothesis list node0);
        fprintf(fp,"address = %d0,q);
        fprintf(fp,"flink = %d flprev = %d0,q->flink,q->flprev);
        fprintf(fp,"flnext = %d clprnt = %d0,
```

```
            q->flnext,q->clprnt);
          q = q->flnext;                          /* advance pointer */
        }
        p = p->hnext;                             /* advance pointer */
      }
      fclose (fp);                                /* close the file */
      system ("list HARDCOPY !lpr");              /* print the file */
      printf("filtered classification hypothesis list node list sent0);
      printf("to the printer with the name of HARDCOPY0); /* user msg */
    }
    return;                                       /* return to point of call */
}

/* This routine sends a copy of the classification hypothesis list nodes
   list to a scratch file and then calls the system automatically to print
   the file */
chlnlhc () {
  struct evnt *p;
  struct chypl *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)  /* open scratch file */
    printf("unable to open scratch file0);
  else {                                          /* if opened */
    p = evhdr->fll;                               /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"classification hypothesis list node list0);
    while (p != evtl) {                           /* while nodes left, do */
      q = (p->hgher)->cnxt;                        /* point to 1st node */
      fprintf(fp,"for event hypothesis at %d0,p);
      while (q != p->chtail) {                    /* while not thru list, do */
        fprintf(fp,"classification hypothesis list node0);
        fprintf(fp,"address = %d0,q);
        fprintf(fp,"hlink = %d cprev = %d0,q->hlink,q->cprev);
        fprintf(fp,"cnxt = %d cparnt = %d0,
          q->cnxt,q->cparnt);
        q = q->cnxt;                              /* advance pointer */
      }
```

```
      p = p->fll;                              /* advance pointer */
    }
    fclose (fp);                               /* close the file */
    system ("list HARDCOPY |lpr");             /* print the file */
    printf("classification hypothesis list node list sent to the0);
    printf("printer with the name of HARDCOPY0); /* user msg */
  }
  return;                                      /* return to point of call */
}

/* This routine sends a copy of the velocity list nodes list to a
   scratch file and then calls the system automatically to print
   the file */
vlnlhc () {
  struct evnt *p;
  struct velist *q;
  if ((fp = fopen(pfilnam,"w")) == NULL)  /* open scratch file */
    printf("unable to open scratch file0);
  else {                                       /* if opened */
    p = evhdr->fll;                            /* set pointer to 1st node */
    fprintf(fp,"at time = %f0,systime);
    fprintf(fp,"velocity list node list0);
    while (p != evtl) {                        /* while nodes left, do */
      q = (p->vhdr)->vnext;                     /* point to 1st node */
      fprintf(fp,"for event hypothesis at %d0,p);
      while (q != p->vsent) {                 /* while not thru list, do */
        fprintf(fp,"velocity list node0);
        fprintf(fp,"address = %d0,q);
        fprintf(fp,"sens1 = %d sens2 = %d vel = %f0,
          q->sens1,q->sens2,q->vel);
        fprintf(fp,"vlast = %d vnext = %d vpar = %d0,
          q->vlast,q->vnext,q->vpar);
        q = q->vnext;                          /* advance pointer */
      }
      p = p->fll;                              /* advance pointer */
    }
```

```
    fclose (fp);                                /* close the file */
    system ("list HARDCOPY |lpr");              /* print the file */
    printf("velocity list node list sent to the printer0);
    printf("with the name of HARDCOPY0); /* user msg */
  }
  return;                                       /* return to point of call */
}

/* This routine sends a copy of the sensor filtered data lists list to a
   scratch file and then calls the system automatically to print
   the file */
sfdllhc () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatafl *r;
  if ((fp = fopen(pfilnam,"w")) == NULL)          /* open scratch file */
    printf("unable to open scratch file0);
  else {                                          /* if opened */
    p = sidhdr->nxt;                              /* set pntr to 1st node */
    fprintf(fp,"at time = %f0,systime);     /* write time to file */
    fprintf(fp,"filtered data list list0);        /* header info */
    while (p != sidtl) {                          /* while nodes left, do */
      q = p->indic;                               /* point to sensor node */
      r = (q->hd)->fdlnx;                      /* pnt to 1st fil data nd */
      fprintf(fp,"for sensor %d0,p->idno);
      while (r != q->tl) {                        /* for all fil data nodes */
        fprintf(fp,"filtered data list node0);
        fprintf(fp,"address = %d0,r);
        fprintf(fp,"fdltyp = %d0,r->fdltyp);
        fprintf(fp,"fdlpr = %d fdlnx = %d0,r->fdlpr,r->fdlnx);
        fprintf(fp,"dfhead = %d dfsent = %d0,r->dfhead,r->dfsent);
        fprintf(fp,"dlprnt = %d0,r->dlprnt);
        r = r->fdlnx;                             /* advance pointer */
      }
      p = p->nxt;                                 /* advance pointer */
    }
```

```
    fclose (fp);                                    /* close the scratch file */
    system ("list HARDCOPY |lpr");                  /* print the file */
    printf("filtered data lists list sent to the printer0); /* user msg */
    printf("with the name of HARDCOPY0);
  }
  return;                                           /* return to point of call */
}
```

```
/* This routine simulates data generation by ground sensors.  It
   keeps a list of targets generated by the user, checks and updates
   target positions, and causes sensor activations to be generated if
   targets are within sensor detection range and all other necessary
   sensor activation criteria are met.  It also allows the user to
   modify, add to or delete from the target list dynamically. */

#include <stdio.h>
#include "pgmvar.c"          /* include program data */
#include "genvar.c"          /* include generator data */
#include "exincld.c"         /* include external variables */

datagen () {
  int chg, nostop;
  char ans;
  chg = FALSE;             /* set change indicator to no change */
  nostop = TRUE;             /* set stopping condition to no stop */
  if ((fp = fopen (dfilnam, "w")) == NULL) /* open file with write access */
    printf("unable to open data file0);/* if it can be opened */
  else {
    fprintf(fp,"%f",systime);    /* write current time to file */
    printf("do you desire a printout of the target list?(y or n)"); /* ask
      user if he wants a target list print */
    ans = getans ();       /* read user answer */
    if (ans == 'y')     /* if response is yes */
      prntlst ();      /* print the target list */
    while (nostop) {    /* do while not stop */
      printf("any changes to the target list?(y or n)"); /* ask if changes */
      ans = getans ();  /* get user response */
      if (ans == 'y') {      /* if answer is yes */
        chg = TRUE;          /* set the change indicator */
        chglist (); }        /* go change the target list */
      else                   /* otherwise */
        nostop = FALSE;      /* set the stopping condition */
    }
    if (chg) {               /* if changes made */
```

```
      printf("do you desire a printout of the target list?(y or n)");
        /* ask user if he wants a target list print */
      ans = getans ();      /* get user response */
      if (ans == 'y')        /* if answer is yes */
        prntlst (); }       /* print the target list */
    gensdat ();             /* generate sensor data */
    fclose (fp);             /* close file */
  }
  return;                    /* return to point of call */
}

/* This routine generates sensor data for the current number of sensors */
gensdat () {
  int i, j, k, bcntr, temp, inotrk;
  struct sensid *p;
  char sdata[MAXTRKS];                  /* sensor data data structure */
  bstim = systime;                      /* set window base time */
  senscnt = countsn ();                 /* determine number of tracks */
  if (senscnt > MAXTRKS)                /* if outside limits */
    printf("tracks required exceeds program limits0); /* error msg */
  bcntr = 1;                          /* set the bit position counter */
  for (k=0;k<MAXTRKS;k++)      /* zero data bytes initially */
    sdata[k] = 0;
  for (i=0;i<(TMWINDW*NPERSEC);i++) { /* do for number of time periods */
    p = sidhdr->nxt;          /* set pointer to first sensor list node */
    j = 0;                    /* set node counter */
    while (p != sidtl) {      /* do for all sensors */
      switch (((p->indic)->type)/10) {  /* case stmt for sensor type */
        case 1000:
          inotrk = gomnsid (j, sdata, p->indic, bcntr); /* minisid s/o */
          break;
        default:
          printf("no data generator for %d0,((p->indic)->type)/10);
            /* print error message */
          sdata[j] =& (~mask[bcntr]); /* put a 0 in that bit position */
          break;
```

```
      }
      p = p->nxt;              /* advance list pointer */
      j =+ inotrk;             /* increment counter to align next sensor */
    }
    if (bcntr == 6) {          /* if a full byte for each sensor */
      for (k=0;k<senscnt;k++) {              /* write out to file */
        if (sdata[k] == 0)                   /* if no activations */
          sdata[k] =| mask[0];               /* set so no null char */
        else                                 /* otherwise */
          sdata[k] =& (~mask[0]);            /* zero first (control) bit */
        fprintf(fp,"%c",sdata[k]);
      }
      bcntr = 1;               /* reset bit counter */
      for (k=0;k<MAXTRKS;k++)      /* zero data bytes initially */
        sdata[k] = 0;
    }
    else                       /* otherwise */
      bcntr =+ 1;              /* increment bit counter */
    movetgt (1.0 / NPERSEC); /* move targets another increment */
    systime =+ (1.0 / NPERSEC);  /* increment system clock */
  }
  if ((temp = TMWINDW * NPERSEC % 6) != 0)   /* odd number of bits ? */
    for (i=0;i<senscnt;i++) {  /* zero remaining bits in last byte */
      for ((j=temp+1);j<7;j++)     /* for all sensors */
        sdata[i] =& (~mask[j]);
      if (sdata[i] == 0)                 /* if no activations */
        sdata[i] =| mask[0];             /* set so no null char */
      else                               /* otherwise */
        sdata[i] =& (~mask[0]);          /* zero first (control) bit */
      fprintf(fp,"%c",sdata[i]);
    }
  return;                      /* return to point of call */
}

/* This routine moves all targets for the specified time interval */
movetgt (timstp)
```

```
  float timstp; {
  int i;
  for (i=0;i<MAXTGTS;i++) {     /* for each target */
    if (tgttype[i] != NONE) {  /* if a target */
      tgtlocx[i] =+ (timstp * tgtvelx[i] / 10.0); /* change x location */
      tgtlocy[i] =+ (timstp * tgtvely[i] / 10.0); /* change y location */
      if ((tgtlocx[i] < MINMAPX) ¦¦  (tgtlocx[i] > MAXMAPX) ¦¦ /* if tgt */
        (tgtlocy[i] < MINMAPY) ¦¦ (tgtlocy[i] > MAXMAPY)) /* off map */
        tgttype[i] = NONE;       /* take target out of list */
    }
  }
  return;                        /* return to point of call */
}

/* This routine prints the target list and target parameters */
prntlst () {
  int gotoit;
  char answ;
  gotoit = TRUE;                /* set initial looping condition */
  while (gotoit) {              /* while not stop */
    printf("options: h(ardcopy list  o(ne by one to terminal0);
    printf("          q(uit0);  /* options list */
    printf("choose one...");
    answ = getans ();         /* get user response */
    switch (answ) {            /* case statement for options */
      case 'h':
        hrdcopy ();           /* target list to hrdcopy file */
        break;
      case 'o':
        oneby1 ();           /* list to terminal one by one */
        break;
      case 'q':
        gotoit = FALSE;        /* exit routine */
        break;
      default:
        printf("command not understood0);  /* print error message */
```

```
        break;
      }
    }
    return;                          /* return to point of call */
}

/* This routine sends a copy of the target list to a scratch file
   and then calls the system automatically to print the file */
hrdcopy () {
  FILE *fphdcpy;
  int i;
  if ((fphdcpy = fopen(pfilnam, "w")) == NULL) /* open file for write */
    printf("unable to open scratch file0);
  else {                             /* if opened, do */
    fprintf(fphdcpy,"at %f0,systime);     /* write current time */
    for (i=0;i<MAXTGTS;i++) {  /* for each target */
      fprintf(fphdcpy,"target %d0ype = %d0,i+1,tgttype[i]);
      fprintf(fphdcpy,"velx = %f vely = %f0,tgtvelx[i],tgtvely[i]);
      fprintf(fphdcpy,"locx = %f locy = %f0,tgtlocx[i],tgtlocy[i]);
    }
    fclose(fphdcpy);                 /* close file */
    system("list HARDCOPY |lpr"); /* send to printer */
    printf("target list transferred to printer with0); /* inform user */
    printf("the name of HARDCOPY0);
  }
  return;                             /* return to point of call */
}

/* This routine sends a copy of the target list to the terminal
   one at a time on the user's command */
oneby1 () {
  int i, temp, keepgo;
  printf("use the carriage return to step through the0); /* give user */
  printf("target file one at a time0);  /* instructions */
  printf("0);                        /* skip a line */
  printf("at %f0,systime);           /* print current time */
```

```
  for (i=0;i<MAXTGTS;i++) {          /* do for all targets */
    keepgo = TRUE;                   /* set looping condition */
    while (keepgo)
      if ((temp = getchar()) == '0') { /* if CR pressed */
        printf("target %d0ype = %d0,i+1,tgttype[i]);
        printf("velx = %f vely = %f0,tgtvelx[i],tgtvely[i]);
        printf("locx = %f locy = %f0,tgtlocx[i],tgtlocy[i]);
        keepgo = FALSE;              /* set stopping condition */
      }
      else                           /* otherwise */
        printf("?0);                 /* print error message */
  }
  return;                             /* return to point of call */
}

/* This routine allows the user to change the target list */
chglist () {
  int i, loopit;
  char answr;
  loopit = TRUE;                     /* set looping condition */
  while (loopit) {                   /* while not stop, do */
    printf("options: a(dd a target  d(elete a target0); /* list user */
    printf("         c(hange tgt param  q(uit0);   /* options */
    printf("choose one...");
    answr = getans ();               /* get user response */
    switch (answr) {                 /* case statement for options */
      case 'a':
        adtolst ();                  /* add a target */
        break;
      case 'd':
        dlfmlst ();                  /* delete a target */
        break;
      case 'c':
        chgparm ();                  /* change a target parameter */
        break;
      case 'q':
```

```
        loopit = FALSE;          /* set condition for exit */
        break;
      default:
        printf("command not understood0);  /* print error message */
        break;
    }
  }
  return;                        /* return to point of call */
}

/* This routine counts the number of sensor tracks to be
   generated for a particular time window */
countsn () {
  struct sensid *p;
  int count;                              /* track counter */
  count = 0;                              /* initialize count */
  p = sidhdr->nxt;                        /* set pointer to first list node */
  while (p != sidtl) {                    /* count all tracks */
    switch (((p->indic)->type)/10) {      /* case stmt for sensor types */
      case 1000:
        count =+ 1;                       /* define 1 track for type */
        chktrk (p->idno,1);               /* check target track list */
        break;
      default:
        count =+ 1;                       /* define 1 track for unrecog type */
        chktrk (p->idno,1);               /* check target track list */
        break;
    }
    p = p->nxt;                           /* advance list pointer */
  }
  dtgtnod ();                             /* delete non-current nodes */
  return (count);                         /* return count to point of call */
}

/* This routine checks the target track list to see if a sensor
   has the proper number of list nodes already allocated.  If not,
```

```
   it allocates the proper number of nodes. */
chktrk (idnum, n)
  int idnum, n; {
  int i;
  struct tgttrk *p, *q;
  p = thead;                              /* set initial pointers */
  q = p->tnext;
  ttail->sensrno = idnum;                 /* set search stop condition */
  while (q->sensrno < idnum) {            /* while not located */
    p = q;                                /* advance pointers */
    q = p->tnext;
  }
  if ((q->sensrno > idnum) || (q == ttail)) /* insert point found */
    intgtnd (n, p, q, idnum);              /* insert required nodes */
  else                                    /* otherwise */
    for (i=0;i<n;i++) {                   /* node(s) have been found */
      q->lstchk = systime;                /* update time last checked */
      q = q->tnext;                       /* move to next affected node */
    }
  return;                                 /* return to point of call */
}

/* This routine inserts the specified number of target list nodes */
intgtnd (n, p, q, id)
  int n, id;
  struct tgttrk *p, *q; {
  int i;
  struct tgttrk *temp;
  for (i=0;i<n;i++) {                  /* for the required nodes */
    temp = (struct tgttrk *) calloc (1,(sizeof (struct tgttrk)));
      /* allocate necessary storage */
    temp->tnext = q;                    /* establish link */
    p->tnext = temp;
    temp->sensrno = id;                 /* establish node initial values */
    temp->nactv = 0;
    temp->lstact = INITTM;
```

```
    temp->lstchk = systime;
    q = temp;                          /* reset pointer if more than 1 */
  }
  return;                              /* return to point of call */
}

/* This routine checks the target track list at the end of counting
   tracks to see which nodes are present for which there are no active
   sensors.  If any are found, they are removed from the list. */
dtgtnod () {
  struct tgttrk *p, *q, *temp;
  p = thead;                           /* set initial pointers */
  q = p->tnext;
  while (q != ttail) {                 /* while not through list */
    if (q->lstchk != systime) {        /* if check not done currently */
      p->tnext = q->tnext;             /* break link */
      temp = q;                        /* set temporary pointer */
      q = temp->tnext;                 /* set lead pointer to next node */
      cfree (temp);                    /* deallocate storage */
    }
    else {                             /* otherwise */
      p = q;                           /* advance pointers */
      q = p->tnext;
    }
  }
  return;                              /* return to point of call */
}

/* This routine allows a user to add to the target list if not full */
adtolst () {
  int nstp, i;
  nstp = TRUE;                         /* initialize loop condition to no stop */
  i = 0;                               /* initialize counter */
  while ((nstp) && (i < MAXTGTS))      /* search for a free location */
    if (tgttype[i] == NONE)            /* if an empty spot found */
      nstp = FALSE;                    /* set stopping condition */
```

```
    else                                /* otherwise */
      i++;                              /* increment counter */
  if (i == MAXTGTS)                     /* if no empty position found */
    printf("insert not possible-list is full0); /* print error msg */
  else {                                /* otherwise */
    typset (i);                         /* set target type */
    velxset (i);                        /* set target X velocity */
    velyset (i);                        /* set target Y velocity */
    locxset (i);                        /* set target X location */
    locyset (i);                        /* set target Y location */
  }
  return;                               /* return to point of call */
}

/* This routine allows the user to delete a target from the target list */
dltmlst () {
  int i;
  i = gettgt ();                        /* get target number from user */
  tgttype[i] = NONE;                    /* delete the target */
  return;                               /* return to point of call */
}

/* This routine allows a user to change a target parameter */
chgparm () {
  int i, nstp;
  char ans;
  nstp = TRUE;                          /* set loop condition to no stop */
  i = gettgt ();                        /* get target number from user */
  while (nstp) {                        /* while no stop, do */
    printf("options: t(ype  x(velocity v(elocity y0); /* print user */
    printf("         l(ocation x       y(location0);    /* options */
    printf("         q(uit0);
    printf("choose one...");
    ans = getans ();                   /* get user response */
    switch (ans) {                      /* case statement for options */
      case 't':
```

```
        typset (i);                     /* change the type */
        break;
      case 'x':
        velxset (i);                    /* change X velocity */
        break;
      case 'v':
        velyset (i);                    /* change Y velocity */
        break;
      case 'l':
        locxset (i);                    /* change X location */
        break;
      case 'y':
        locyset (i);                    /* change Y location */
        break;
      case 'q':
        nstp = FALSE;                   /* set stopping condition to exit */
        break;
      default:
        printf("command not understood0); /* print error msg */
        break;
    }
  }
  return;                               /* return to point of call */
}

/* This routine sets the target type given its array location */
typset (i)
  int i; {
  int tempi, nstp;
  nstp = TRUE;                          /* set loop condition to no stop */
  while (nstp) {                        /* while not stop, do */
    printf("enter type");               /* print user instructions */
    tempi = getint ();              /* convert to integer */
    if ((tempi < 1) || (tempi > NTTYP)) /* if out of type range */
      printf("type description out of bounds0); /* print error msg */
    else {                              /* otherwise */
```

```
      tgttype[i] = tempi;          /* accept target type */
      nstp = FALSE;                /* set stopping condition */
    }
  }
  return;                          /* return to point of call */
}

/* This routine sets the target velocity in the X direction */
velxset (i)
  int i; {
  printf("a negative velocity indicates movement to left (west)0);
  printf("enter velocity (m/sec) in the X direction"); /* user instr */
  tgtvelx[i] = gtfloat (); /* convert to floating point, store */
  return;                          /* return to point of call */
}

/* This routine sets the target velocity in the Y direction */
velyset (i)
  int i; {
  printf("a negative velocity indicates movement down (south)0);
  printf("enter velocity (m/sec) in the Y direction"); /* user instr */
  tgtvely[i] = gtfloat ();  /* convert to floating point, store */
  return;                          /* return to point of call */
}

/* This routine sets the target location in the X direction */
locxset (i)
  int i; {
  int nstp;
  float tempf;
  nstp = TRUE;                      /* set loop condition to no stop */
  while (nstp) {                    /* while not stop, do */
    printf("enter X UTM coord (0000-1000)"); /* print user instructions */
    tempf = gtfloat (); /* convert to floating point */
    if ((tempf < MINMAPX) || (tempf > MAXMAPX)) /* if out of bounds */
      printf("X coord out of bounds0);   /* print error msg */
```

```
    else {                              /* otherwise */
      tgtlocx[i] = tempf;               /* accept X coord */
      nstp = FALSE;                     /* set stopping condition */
    }
  }
  return;                               /* return to point of call */
}

/* This routine sets the target location in the Y direction */
locyset (i)
  int i; {
  int nstp;
  float tempf;
  nstp = TRUE;                          /* set loop condition to no stop */
  while (nstp) {                        /* while not stop, do */
    printf("enter Y UTM coord (0000-1000)"); /* print user instructions */
    tempf = gtfloat ();  /* convert to floating point, store */
    if ((tempf < MINMAPY) || (tempf > MAXMAPY)) /* if out of bounds */
      printf("Y coord out of bounds0); /* print error msg */
    else {                              /* otherwise */
      tgtlocy[i] = tempf;               /* accept Y target location */
      nstp = FALSE;                     /* set stopping condition */
    }
  }
  return;                               /* return to point of call */
}

/* This routine gets the target number to be used from the user */
gettgt () {
  int i, nostp;
  nostp = TRUE;                         /* set loop condition to no stop */
  while (nostp) {                       /* while not stop, do */
    printf("target number 1 is first in the list.0); /* print user */
    printf("enter target number of the target"); /* instructions */
    i = getint ();                     /* convert to integer */
    if ((i < 1) || (i > MAXTGTS))       /* if out of bounds */
```

```
      printf("target number is out of bounds0); /* print error msg */
    else                                /* otherwise */
      if (tgttype[i-1] == NONE)         /* if a target is not there */
        printf("no target there0);      /* print error msg */
      else                              /* otherwise */
        nostp = FALSE;                  /* set stopping condition */
  }
  return (i-1);                         /* return index to pont of call */
}

/* This routine generates data for a minisid sensor in the seismic only
   configuration */
gomnsid (j, s, q, bcntr)
  int j, bcntr;
  char s[];
  struct sensnod *q; {
  double x, y;
  struct tgttrk *p;
  int i, nstp;
  nstp = TRUE;                                  /* set loop condition to no stop */
  p = thead->tnext;                             /* set pointer to first node */
  for (i=0;i<j;i++)                             /* do until at right node */
    p = p->tnext;                               /* advance pointer */
  s[j] =& (~mask[bcntr]);                       /* zero bit initially */
  if (((systime-(p->lstact)) >= q->hs) &&       /* if sensor can be activated */
    (q->tmactiv <= systime))
    for (i=0;i<MAXTGTS;i++)                     /* do for all targets */
      if (nstp)                                 /* while no activation */
        if (tgttype[i] != NONE) {               /* if a target is there */
          x = (double) ((tgtlocx[i]-q->slocx)*10.0); /* check if target */
          y = (double) ((tgtlocy[i]-q->slocy)*10.0); /* within DR */
          if ((float) (sqrt(pow(x,((double) (2.0))) + (pow(y,((double)
            (2.0)))))) <= q->detrad[(tgttype[i]-1)]) {
            nstp = FALSE;                       /* set stopping condition */
            s[j] =! mask[bcntr];                /* cause an activation */
            if (((systime-(p->lstact))>=((q->hs)-FFAC))&&((systime-
```

```
                (p->lstact))<=((q->hs)+FFAC))) /* if activation continuous */
                p->nactv =+ 1;                 /* increment counter */
              else                             /* otherwise */
                p->nactv = 1;                  /* reset counter */
              p->lstact = systime;             /* set time of activation */
            }
          }
  return (1);                                  /* return number of tracks used */
}
```

```
/* This module contains the production rules used by the AI ground
   sensor program */

#include <stdio.h>
#include "pgmvar.c"                          /* include program data */
#include "exincld.c"                         /* include external variables */

/* This routine is a production rule level 0 to level 1 */
l011r0 () {
  struct arayid *p;
  struct araynod *q;
  struct sensnod *r, *s;
  float A, val, tag;
  int cntr;
  p = aidhdr->next;                          /* point to 1st list node */
  while (p != aidtl) {                       /* while not thru list, do */
    q = p->pntr;                             /* point to array node */
    r = (q->headr)->nx;                      /* point to 1st sensor node */
    while (r != q->sntinl) {                 /* while not thru list, do */
      if (r->hyprsen >= 0.0)                 /* if sensor not hypersenstive */
        if ((systime - r->hyprsen) >= ((float)(TMWINDW * NWINDW))) {
          /* if sensor has been this way >= required time */
          cntr = 0;                          /* initialize activation cntr */
          val = 0.0;                         /* initialize expectation sum */
          s = (q->headr)->nx;                /* point to 1st node */
          switch (r->type/10) {              /* case stmt for sensor type */
            case 1000:                       /* for minisid seismic only */
              A = 7.5;                       /* set expectation limit */
              break;
            default:                         /* for all other types */
              A = 7.5;                       /* set expectation limit */
              break;
          }
          while (s != q->sntinl) {           /* while not at end of list, do */
            if (r != s) {                    /* if not on sensor being tested */
              switch (s->type/10) {          /* case stmt for type */
```

```
                case 1000:                  /* for minisid seismic only */
                  if ((s->hyprsen >= 0.0) && ((systime - s->hyprsen) >=
                    ((float)(IMWINDW * NWINDW))) && (s->offon >= 0.0)) {
                    /* if sensor should be included in calculations */
                    cntr =+ 1;              /* increment counter */
                    val =+ s->expect;       /* increment expectation sum */
                  }
                  break;
                default:                    /* for all other types */
                  break;                    /* ignore */
              }
            }
            s = s->nx;                      /* advance pointer */
          }
          if (cntr > 0) {                   /* if comparison possible */
            tag = (1.0 / (float)cntr) * val; /* get average */
            switch (r->type/10) {           /* case stmt for sensor type */
              case 1000:                    /* for minisid seismic only */
                if ((r->expect > A) && (r->expect > tag)) /* if hypsen */
                  r->hyprsen = -systime;    /* set to hypersensitive */
                break;
              default:                      /* for all other types */
                break;                      /* ignore */
            }
          }
        }
        r = r->nx;                          /* advance pointer */
      }
      p = p->next;                          /* advance pointer */
    }
    return;                                 /* return to point of call */
}

/* This routine is a production rule level 0 to level 1 */
1011r1 () {
  struct arayid *p;
```

```
struct araynod *q;
struct sensnod *r, *s;
float A, val, tag;
int cntr;
p = aidhdr->next;                          /* point to 1st list node */
while (p != aidtl) {                       /* while not thru list, do */
  q = p->pntr;                             /* point to array node */
  r = (q->headr)->nx;                      /* point to 1st sensor node */
  while (r != q->sntinl) {                 /* while not thru list, do */
    if (r->hyprsen < 0.0)                  /* if sensor hypersenstive */
      if ((systime + r->hyprsen) >= ((float)(TMWINDW * NWINDW))) {
        /* if sensor has been this way >= required time */
        cntr = 0;                          /* initialize activation cntr */
        val = 0.0;                         /* initialize expectation sum */
        s = (q->headr)->nx;                /* point to 1st node */
        switch (r->type/10) {              /* case stmt for sensor type */
          case 1000:                       /* for minisid seismic only */
            A = 7.5;                       /* set expectation limit */
            break;
          default:                         /* for all other types */
            A = 7.5;                       /* set expectation limit */
            break;
        }
        while (s != q->sntinl) {           /* while not at end of list, do */
          if (r != s) {                    /* if not on sensor being tested */
            switch (s->type/10) {          /* case stmt for type */
              case 1000:                   /* for minisid seismic only */
                if ((s->hyprsen >= 0.0) && ((systime - s->hyprsen) >=
                  ((float)(TMWINDW * NWINDW))) && (s->offon >= 0.0)) {
                  /* if sensor should be included in calculations */
                  cntr =+ 1;               /* increment counter */
                  val =+ s->expect;        /* increment expectation sum */
                }
                break;
              default:                     /* for all other types */
                break;                     /* ignore */
```

```
            }
          }
          s = s->nx;                          /* advance pointer */
        }
        if (cntr > 0) {                       /* if comparison possible */
          tag = (1.0 / (float)cntr) * val; /* get average */
          switch (r->type/10) {               /* case stmt for sensor type */
            case 1000:                        /* for minisid seismic only */
              if ((r->expect <= A) || (r->expect <= tag))
                /* if not hypersensitive */
                r->hyprsen = systime;         /* set to not hypersensitive */
              break;
            default:                          /* for all other types */
              break;                          /* ignore */
          }
        }
      }
      r = r->nx;                          /* advance pointer */
    }
    p = p->next;                              /* advance pointer */
  }
  return;                                     /* return to point of call */
}

/* This routine is a production rule level 0 to level 1 */
1011r2 () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatau *r;
  p = sidhdr->nxt;                            /* point to first list node */
  while (p != sidtl) {                        /* while not thru list, do */
    q = p->indic;                             /* point to sensor node */
    if (q->offon < 0.0) {                     /* if sensor is inactive */
      r = (q->head)->right;                   /* point to 1st uf data node */
      if (r != q->tail)                       /* if list not empty */
        q->offon = r->timact;                 /* make sensor active */
```

```
    }
    p = p->nxt;                                  /* advance pointer */
  }
  return;                                        /* return to point of call */
}

/* This routine is a production rule level 0 to level 1 */
l011r3 () {
  struct arayid *p;
  struct araynod *q;
  struct sensnod *r, *s;
  float B, val, tag;
  int cntr;
  p = aidhdr->next;                              /* point to 1st list node */
  while (p != aidtl) {                           /* while not thru list, do */
    q = p->pntr;                                 /* point to array node */
    r = (q->headr)->nx;                          /* point to 1st sensor node */
    while (r != q->sntinl) {                     /* while not thru list, do */
      if (r->offon >= 0.0)                     /* if sensor not inactive */
        if ((systime - r->offon) >= ((float)(TMWINDW * NWINDW))) {
          /* if sensor has been this way >= required time */
          cntr = 0;                              /* initialize activation cntr */
          val = 0.0;                             /* initialize expectation sum */
          s = (q->headr)->nx;                    /* point to 1st node */
          switch (r->type/10) {                  /* case stmt for sensor type */
            case 1000:                           /* for minisid seismic only */
              B = 0.6;                           /* set expectation limit */
              break;
            default:                             /* for all other types */
              B = 0.6;                           /* set expectation limit */
              break;
          }
          while (s != q->sntinl) {               /* while not at end of list, do */
            if (r != s) {                        /* if not on sensor being tested */
              switch (s->type/10) {              /* case stmt for type */
                case 1000:                       /* for minisid seismic only */
```

```
                    if ((s->offon >= 0.0) && ((systime - s->offon) >=
                      ((float)(IMWINDW * NWINDW))) && (s->hyprsen >= 0.0)) {
                      /* if sensor should be included in calculations */
                      cntr =+ 1;              /* increment counter */
                      val =+ s->expect;       /* increment expectation sum */
                    }
                    break;
                  default:                    /* for all other types */
                    break;                    /* ignore */
                }
              }
              s = s->nx;                      /* advance pointer */
            }
            if (cntr > 0) {                   /* if comparison possible */
              tag = (1.0 / (float)cntr) * val; /* get average */
              switch (r->type/10) {           /* case stmt for sensor type */
                case 1000:                    /* for minisid seismic only */
                  if ((r->expect < B) && (r->expect < tag)) /* if inactv */
                    r->offon = -systime;      /* set to inactive */
                  break;
                default:                      /* for all other types */
                  break;                      /* ignore */
              }
            }
          }
        r = r->nx;                            /* advance pointer */
      }
      p = p->next;                            /* advance pointer */
    }
    return;                                   /* return to point of call */
}

/* This routine is a production rule level 1 to level 2 */
1112r0 () {
  struct sensid *p;
  struct sensnod *q;
```

```
  struct sdatau *r;
  struct sdataf1 *s, *cfsdnln (), *sfsdnln ();
  struct sdataf *t, *cfsdn ();
  p = sidhdr->nxt;                          /* point to 1st list node */
  while (p != sidtl) {                      /* while not thru list, do */
    q = p->indic;                           /* point to sensor node */
    switch (q->type/10) {                   /* case stmt for type */
      case 1000:                            /* for minisid seismic only */
        r = (q->head)->right;               /* point to 1st uf data node */
        if (r != q->tail) {                 /* if activations in this prd */
          s = (q->hd)->fdlnx;                 /* point to 1st list node */
          if (s == q->tl) {                   /* if no list nodes */
            s = cfsdnln (q);                  /* create and insert new nod */
            s->fdltyp = 0;                    /* set type of filtering */
          }
          else {                              /* otherwise */
            s = sfsdnln (q, 0);               /* search for same type node */
            if (s == NIL) {                   /* if not present */
              s = cfsdnln (q);                /* create one */
              s->fdltyp = 0;                  /* set type of filtering */
            }
          }
          while (r != q->tail) {              /* while not thru list, do */
            t = cfsdn (s);                    /* create a filtered data node */
            t->tmact = r->timact;             /* set activation time */
            r = r->right;                     /* advance pointer */
          }
        }
        break;
      default:                              /* for all other types */
        break;
    }
    p = p->nxt;                             /* advance pointer */
  }
  return;                                   /* return to point of call */
}
```

```
/* This routine is a production rule level 1 to level 2 */
l1l2r1 () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatau *r;
  struct sdataf1 *s, *cfsdnln (), *sfsdnln ();
  struct sdataf *t, *cfsdn ();
  p = sidhdr->nxt;                          /* point to 1st list node */
  while (p != sidtl) {                      /* while not thru list, do */
    q = p->indic;                           /* point to sensor node */
    switch (q->type/10) {                   /* case stmt for type */
      case 1000:                            /* for minisid seismic only */
        r = (q->head)->right;               /* point to 1st uf data node */
        if (r != q->tail)                   /* if activations in this prd */
          if (q->hyprsen >= 0.0) {          /* if sensor not hypersensitive */
            s = (q->hd)->fdlnx;                 /* point to 1st list node */
            if (s == q->tl) {                   /* if no list nodes */
              s = cfsdnln (q);                  /* create and insert new nod */
              s->fdltyp = 1;                    /* set type of filtering */
            }
            else {                              /* otherwise */
              s = sfsdnln (q, 1);               /* search for same type node */
              if (s == NIL) {                   /* if not present */
                s = cfsdnln (q);                /* create one */
                s->fdltyp = 1;                  /* set type of filtering */
              }
            }
            while (r != q->tail) {              /* while not thru list, do */
              t = cfsdn (s);                    /* create a filtered data node */
              t->tmact = r->timact;             /* set activation time */
              r = r->right;                     /* advance pointer */
            }
          }
        break;
      default:                              /* for all other types */
```

```
        break;
      }
      p = p->nxt;                                   /* advance pointer */
    }
    return;                                         /* return to point of call */
}

/* This routine is a production rule level 2 to level 3 */
1213r0 () {
  struct sensid *p;
  struct sensnod *q;
  struct sdataf1 *r;
  struct sdataf *s, *srfldl ();
  struct vset *t, *vstofdn (), *crtvset ();
  float gtcntim ();
  int temp;
  p = sidhdr->nxt;                                  /* point to 1st list node */
  while (p != sidtl) {                              /* while not thru list, do */
    q = p->indic;                                   /* point to sensor node */
    switch (q->type/10) {                           /* case stmt for snsr type */
      case 1000:                                    /* for minisid seismic only */
        r = (q->hd)->fdlnx;                         /* point to 1st list node */
        while (r != q->tl) {                        /* while not thru list, do */
          s = srfldl (r);                           /* current activations ? */
          if (s != NIL) {                           /* if found */
            t = vstofdn (s);                        /* search valid set list */
            if (t == NIL)                           /* if no match found */
              if ((temp = vscompl (s)) == YES) {    /* if complete set */
                t = crtvset (q);                    /* create a new valid set */
                t->numact = gtnmact (s);            /* get no. activations */
                t->centrd = gtcntim (s);            /* get centroid time */
                t->timupd = systime;                /* set update time */
                t->compl = -YES;                    /* set compl/type code */
                t->1st = s;                         /* link to 1st fil datum */
              }
          }
```

```
            r = r->fdlnx;                          /* advnc fil data ls ls ptr */
          }
          break;
        default:                                   /* for all other types */
          break;                                   /* ignore */
      }
      p = p->nxt;                                  /* advance list pointer */
    }
    return;                                        /* return to point of call */
}

/* This routine is a production rule level 2 to level 3 */
1213r1 () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatafl *r;
  struct sdataf *s, *srfldl ();
  struct vset *t, *vstofdn (), *crtvset ();
  float gtcntim ();
  int temp;
  p = sidhdr->nxt;                                 /* point to 1st list node */
  while (p != sidtl) {                             /* while not thru list, do */
    q = p->indic;                                  /* point to sensor node */
    switch (q->type/10) {                          /* case stmt for snsr type */
      case 1000:                                   /* for minisid seismic only */
        r = (q->hd)->fdlnx;                        /* point to 1st list node */
        while (r != q->tl) {                       /* while not thru list, do */
          s = srfldl (r);                          /* current activations ? */
          if (s != NIL) {                          /* if found */
            t = vstofdn (s);                       /* search valid set list */
            if (t == NIL)                          /* if no match found */
              if ((temp = vscompl (s)) == NO) { /* if not a complete set */
                t = crtvset (q);                   /* create a new valid set */
                t->numact = gtnmact (s);           /* get no. activations */
                t->centrd = gtcntim (s);           /* get centroid time */
                t->timupd = systime;               /* set update time */
```

```
                t->compl = -NO;                    /* set compl/type code */
                t->lst = s;                        /* link to 1st fil datum */
              }
            }
            r = r->fdlnx;                          /* advnc fil data ls ls ptr */
          }
          break;
        default:                                   /* for all other types */
          break;                                   /* ignore */
      }
      p = p->nxt;                                  /* advance list pointer */
    }
    return;                                        /* return to point of call */
}

/* This routine is a production rule level 2 to level 3 */
1213r2 () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatafl *r;
  struct sdataf *s, *srfldl ();
  struct vset *t, *vstofdn ();
  float qtcntim ();
  int temp;
  p = sidhdr->nxt;                                 /* point to 1st list node */
  while (p != sidtl) {                             /* while not thru list, do */
    q = p->indic;                                  /* point to sensor node */
    switch (q->type/10) {                          /* case stmt for snsr type */
      case 1000:                                   /* for minisid seismic only */
        r = (q->hd)->fdlnx;                        /* point to 1st list node */
        while (r != q->tl) {                       /* while not thru list, do */
          s = srfldl (r);                          /* current activations ? */
          if (s != NIL) {                          /* if found */
            t = vstofdn (s);                       /* search valid set list */
            if (t != NIL)                          /* if match found */
              if ((temp = vscompl (s)) == YES) { /* if complete set */
```

```
                t->numact =+ gtnmact (s);       /* get no. activations */
                t->centrd = gtcntim (t->lst);     /* get centroid time */
                t->timupd = systime;            /* set update time */
                t->compl = YES;                 /* set compl/type code */
              }
            }
            r = r->fdlnx;                       /* advnc fil data ls ls ptr */
          }
          break;
        default:                                /* for all other types */
          break;                                /* ignore */
      }
      p = p->nxt;                               /* advance list pointer */
    }
    return;                                     /* return to point of call */
}

/* This routine is a production rule level 2 to level 3 */
1213r3 () {
  struct sensid *p;
  struct sensnod *q;
  struct sdatafl *r;
  struct sdataf *s, *srfldl ();
  struct vset *t, *vstofdn ();
  float gtcntim ();
  int temp;
  p = sidhdr->nxt;                              /* point to 1st list node */
  while (p != sidtl) {                          /* while not thru list, do */
    q = p->indic;                               /* point to sensor node */
    switch (q->type/10) {                       /* case stmt for snsr type */
      case 1000:                                /* for minisid seismic only */
        r = (q->hd)->fdlnx;                     /* point to 1st list node */
        while (r != q->tl) {                    /* while not thru list, do */
          s = srfldl (r);                       /* current activations ? */
          if (s != NIL) {                       /* if found */
            t = vstofdn (s);                    /* search valid set list */
```

```
            if (t != NIL)                          /* if match found */
              if ((temp = vscompl (s)) == NO) { /* if not a complete set */
                t->numact =+ gtnmact (s);      /* get no. activations */
                t->centrd = gtcntim (t->lst);  /* get centroid time */
                t->timupd = systime;           /* set update time */
                t->compl = NO;                 /* set compl/type code */
              }
            }
            r = r->fdlnx;                          /* advnc fil data ls ls ptr */
          }
          break;
        default:                                   /* for all other types */
          break;                                   /* ignore */
      }
      p = p->nxt;                                  /* advance list pointer */
    }
    return;                                        /* return to point of call */
}

/* This routine is a production rule level 3 to level 3 */
1313r0 () {
  struct sensid *p;
  struct sensnod *q;
  struct vset *r;
  p = sidhdr->nxt;                          /* point to 1st list node */
  while (p != sidtl) {                      /* while not thru list, do */
    q = p->indic;                           /* point to sensor node */
    r = (q->hdr)->foll;                     /* point to 1st valid set */
    while (r != q->sentl) {                 /* while not thru list, do */
      if ((r->timupd != systime) && (r->compl != YES)) {
        /* if not changed this cycle and it needs to be */
        r->compl = YES;                     /* then mark valid set complete */
        r->timupd = datatim;                /* as of last cycle */
      }
      r = r->foll;                          /* advance pointer */
    }
```

```
    p = p->nxt;                                  /* advance pointer */
  }
  return;                                        /* return to point of call */
}

/* This routine is a production rule level 3 to level 3 */
1313r1 () {
  struct aravid *p;
  struct araynod *q;
  struct sensnod *r, *s;
  struct vset *t;
  struct evntl *u;
  float maxval, dtim, minval, x, y;
  int ajcncy [NPERMAP], i;
  p = aidhdr->next;                              /* point to 1st list node */
  while (p != aidtl) {                           /* while not thru list, do */
    q = p->pntr;                                 /* point to array node */
    minval = getmin (q);                         /* get min target vel */
    r = (q->headr)->nx;                          /* point to 1st sensor */
    while (r != q->sntinl) {                    /* while not thru list, do */
      gtadsen (r->sbmapno,q,ajcncy);             /* get adjacent sensors */
      if (ajcncy[0] >= (NUMT - 1)) {             /* if ok for comparison */
        t = (r->hdr)->foll;                      /* point to 1st valid set */
        while (t != r->sentl) {                  /* while not thru list, do */
          maxval = 0.0;                          /* initialize maximum value */
          for (i=0;i<ajcncy[0];i++) {            /* for each adjcnt sensr, do */
            s = (q->headr)->nx;                  /* find adjacent sensor */
            while (ajcncy[i+1] != s->sbmapno)
              s = s->nx;
            x = (double)((s->slocx - r->slocx) * 10.0);
            y = (double)((s->slocy - r->slocy) * 10.0);
            dtim = (((float)(sqrt(pow(x,((double)(2.0)))+pow(y,((double)
              (2.0)))))) * 1.1 / minval);
            if (dtim > maxval)                   /* if new max value */
              maxval = dtim;                     /* save it */
          }
```

```
            if (((systime - t->centrd) > maxval) &&  /* if valid set */
              ((u = (t->higher)->follow) == t->eltail)) /* not current */
               rmvset (t);                          /* remove the valid set */
            t = t->foll;                            /* advance pointer */
          }
        }
        r = r->nx;                                  /* advance pointer */
      }
      p = p->next;                                  /* advance pointer */
    }
    return;                                         /* return to point of call */
}

/* This routine is a production rule level 3 to level 4 */
1314r0 () {
  struct sensnod *sptr[NPERMAP], *stemp;
  struct vset *ptr[NPERMAP];
  struct arayid *p;
  struct araynod *q;
  int i, j, k, count, counter, m, savval, ajcnt[NPERMAP], snsrcnt;
  p = aidhdr->next;                                 /* point to 1st list node */
  while (p != aidtl) {                              /* while not thru list, do */
    q = p->pntr;                                    /* point to array */
    if ((q->typ/10) == 1) {                         /* if a trail array */
      j = gtbsnsr (q);                              /* get a boundary sensor */
      if (j >= 0) {                                 /* if no error */
        gtadsen (j, q, ajcnt);                      /* get its adjacent sensors */
        if (ajcnt[0] != 1) {                        /* if a setup error */
          printf("boundary sensor setup for0); /* print error msg */
          printf("array %d is in error0,(q->aidback)->araynum);
          printf("1314r0 not applicable0);
        }
        else {                                       /* otherwise */
          snsrcnt = 0;                              /* set active snsr cnt */
          savval = -1;                              /* initialize check value */
          k = 0;                                    /* initialize index */
```

```
for (i=0;i<q->numsen;i++) {          /* for all array sensrs, do */
  stemp = (q->headr)->nx;            /* point to 1st sensor */
  while (stemp->sbmapno != j)        /* while id'd sensor not fnd */
    stemp = stemp->nx;               /* advance pointer */
  if ((stemp->tmactiv >= 0.0) &&
    (stemp->hyprsen >= 0.0)) {       /* if active and not hyprsen */
    switch (stemp->type/10) {        /* case stmt for sensor type */
      case 1000:                     /* for minisid s/o */
        sptr[k++] = stemp;           /* load pointer array */
        snsrcnt =+ 1;                /* increment sensor count */
        break;
      default:                       /* for all other types */
        break;                       /* ignore */
    }
  }
  j = ((savval == ajcnt[1]) ? ajcnt[2] : ajcnt[1]);
    /* get next sensor in line */
  savval = stemp->sbmapno;           /* new check value */
  gtadsen (j, q, ajcnt);             /* get adjacent sensors */
}
if (snsrcnt >= NUMT)                 /* if required no. of snsrs */
  for (i=0;i<2;i++) {                /* do twice */
    count = 0;                       /* set event hypoth count */
    k = 0;                           /* set level indicator */
    while ((k+NUMT) <= snsrcnt) {/* for required number, do */
      for (j=k;j<snsrcnt;j++)        /* for each sensor */
        ptr[j] = (sptr[j]->hdr)->foll; /* pnt to 1st valid set */
      m = k;                         /* set 1st level indicator */
      counter = 0;                   /* set event hypoth ret cntr */
      counter = search (k,m,sptr,ptr,snsrcnt,q);
        /* start search for hypotheses */
      count =+ counter;              /* increment count */
      k =+ 1;                        /* move to next level */
    }
    m = snsrcnt - 1;                 /* point to last array val */
    for (j=0;j<(snsrcnt/2);j++) {    /* switch order */
```

```
                stemp = sptr[j];                    /* to check from other dir */
                sptr[j] = sptr[m];
                sptr[m] = stemp;
                m -= 1;
              }
            }
          }
        }
      }
      p = p->next;                                  /* advance pointer */
    }
    return;                                         /* return to point of call */
}

/* This routine is a production rule level 4 to level 4 */
1414r0 () {
  struct evnt *p, *q, *r, *temp;
  int retval;
  evtl->updtim = systime;                           /* set absolute stopping cond */
  p = evhdr->fll;                                   /* point to 1st node */
  while (p->updtim != systime)                      /* while not to 1st prsnt hyp */
    p = p->fll;                                     /* advance pointer */
  if (p == evtl)                                    /* if no formed at prsnt time */
    return;                                         /* return - no comparisons */
  else {                                            /* otherwise */
    r = evhdr->fll;                                 /* point to 1st node */
    if (r == p)                                     /* if no past hypotheses */
      return;                                       /* return - no comparisons */
    else {                                          /* otherwise */
      q = p;                                        /* set working pointer */
      while (r != p) {                              /* while not thru past hyp, do */
        while (q != evtl) {                         /* while not thru prsnt set */
          if ((retval = dupehy (r,q)) == YES) {     /* if duplicate nodes */
            if (p == q)                             /* if beginning presnt node */
              p = p->fll;                           /* set new beginning node */
            temp = q;                               /* set temporary pointer */
```

```
            q = q->last;                          /* reset working pointer */
            remehyp (temp);                       /* remove the duplicate node */
          }
          q = q->fll;                             /* advance the pointer */
        }
        q = p;                                    /* reset for next iteration */
        r = r->fll;                               /* advance pointer */
      }
    }
  }
  return;                                         /* return to point of call */
}

/* This routine is a production rule level 4 to level 4 */
1414r1 () {
  struct evnt *p, *q, *r, *temp;
  int retval;
  evtl->updtim = systime;                         /* set absolute stopping cond */
  p = evhdr->fll;                                 /* point to 1st node */
  while (p->updtim != systime)                    /* while not to 1st prsnt hyp */
    p = p->fll;                                   /* advance pointer */
  if (p == evtl)                                  /* if no formed at prsnt time */
    return;                                       /* return - no comparisons */
  else {                                          /* otherwise */
    r = evhdr->fll;                               /* point to 1st node */
    q = p;                                        /* set working pointer */
    while (r != p) {                              /* while not thru past hyp, do */
      while (q != evtl) {                         /* while not thru prsnt hyp */
        if ((retval = ssetehy (q,r)) == YES) { /* if duplicate nodes */
          if (p == q)                             /* if node to be rmvd is 1st */
            p = p->fll;                           /* advance header node */
          temp = q;                               /* set temporary pointer */
          q = q->last;                            /* reset working pointer */
          remehyp (temp);                         /* remove the duplicate node */
        }
        q = q->fll;                               /* advance working pointer */
```

```
        }
        r = r->fll;                          /* advance list pointer */
        q = p;                               /* reset working pointer */
      }
      r = evhdr->fll;                        /* point to 1st node */
      q = p;                                /* set working pointer */
      while (q != evtl) {                   /* while not thru past hyp, do */
        while (r != p) {                    /* while not thru prsnt set */
          if ((retval = ssetehy (r,q)) == YES) { /* if duplicate nodes */
            temp = r;                       /* set temporary pointer */
            r = r->last;                    /* reset working pointer */
            remehyp (temp);                 /* remove the duplicate node */
          }
          r = r->fll;                       /* advance the pointer */
        }
        r = evhdr->fll;                     /* reset for next iteration */
        q = q->fll;                         /* advance pointer */
      }
      q = p->fll;                          /* reset working pointer */
      while (p != evtl) {                  /* while not at list end */
        while (q != evtl) {                /* while not at list end */
          if ((retval = ssetehy (q,p)) == YES) { /* if a subset */
            temp = q;                      /* set temporary pointer */
            q = q->last;                   /* reset working pointer */
            remehyp (temp);                /* remove the subset node */
          }
          q = q->fll;                      /* advance the pointer */
        }
        p = p->fll;                        /* advance base pointer */
        if (p != evtl)                     /* if not finished */
          q = p->fll;                      /* reset working pointer */
      }
    }
    return;                                /* return to point of call */
}
```

```
/* This routine is a production rule level 4 to level 4 */
1414r2 () {
  struct evnt *p, *q, *r, *temp;
  int retval;
  evtl->updtim = systime;                    /* set absolute stopping cond */
  p = evhdr->fll;                            /* point to 1st node */
  while (p->updtim != systime)               /* while not to 1st prsnt hyp */
    p = p->fll;                              /* advance pointer */
  if (p == evtl)                             /* if no formed at prsnt time */
    return;                                  /* return - no comparisons */
  else {                                     /* otherwise */
    r = evhdr->fll;                          /* point to 1st node */
    if (r == p)                              /* if no past hypotheses */
      return;                                /* return - no comparisons */
    else {                                   /* otherwise */
      q = p;                                 /* set working pointer */
      while (q != evtl) {                    /* while not thru prsnt set */
        while (r != p) {                     /* while not thru past set */
          if ((retval = sextehy (r,q)) == YES) { /* if duplicate nodes */
            temp = r;                        /* set temporary pointer */
            r = r->last;                     /* reset working pointer */
            remehyp (temp);                  /* remove the duplicate node */
          }
          r = r->fll;                        /* advance the pointer */
        }
        q = q->fll;                          /* reset for next iteration */
        r = evhdr->fll;                      /* advance pointer */
      }
    }
  }
  return;                                    /* return to point of call */
}

/* This routine is a production rule level 4 to level 5 */
1415r0 () {
  struct evnt *p;
```

```
  struct velist *q;
  struct araynod *r;
  int indic, nostop;
  p = evhdr->fll;                                  /* point to 1st event node */
  while (p != evtl) {                              /* while not thru list, do */
    if (p->updtim == systime) {                    /* if a current form or update */
      nostop = TRUE;                                 /* set no stop indicator */
      indic = NO;                                    /* set class hyp form marker */
      r = ((((p->lowr)->lfoll)->vslink)->lnk)->parent; /* get array ptr */
      if ((p->velmin < r->vmin[0]) && (p->velmax > r->vmax[0])) {
        /* if a classification hypothesis can be formed outright */
        nostop = FALSE;                              /* set stopping condition */
        indic = YES;                                 /* set formation indicator */
      }
      else                                           /* otherwise */
        q = (p->vhdr)->vnext;                        /* prepare list search */
      while ((nostop) && (q != p->vsent)) {  /* while not stop, do */
        if ((q->vel >= r->vmin[0]) && (q->vel <= r->vmax[0])) {
          /* if a classification hypothesis can be formed on velocity */
          nostop = FALSE;                            /* set stopping condition */
          indic = YES;                               /* set formation indicator */
        }
        q = q->vnext;                                /* advance the pointer */
      }
      if (indic == YES)                              /* if a hypoth can be formed */
        csclhyp (p, r, 0);                           /* create hyp and set params */
    }
    p = p->fll;                                    /* advance the pointer */
  }
  return;                                          /* return to point of call */
}

/* This routine is a production rule level 4 to level 5 */
1415r1 () {
  struct evnt *p;
  struct velist *q;
```

```
  struct araynod *r;
  int indic, nostop;
  p = evhdr->fll;                              /* point to 1st event node */
  while (p != evtl) {                          /* while not thru list, do */
    if (p->updtim == systime) {                /* if a current form or update */
      nostop = TRUE;                             /* set no stop indicator */
      indic = NO;                                /* set class hyp form marker */
      r = ((((p->lowr)->lfoll)->vslink)->lnk)->parent; /* get array ptr */
      if ((p->velmin < r->vmin[1]) && (p->velmax > r->vmax[1])) {
        /* if a classification hypothesis can be formed outright */
        nostop = FALSE;                          /* set stopping condition */
        indic = YES;                             /* set formation indicator */
      }
      else                                       /* otherwise */
        q = (p->vhdr)->vnext;                    /* prepare list search */
      while ((nostop) && (q != p->vsent)) {  /* while not stop, do */
        if ((q->vel >= r->vmin[1]) && (q->vel <= r->vmax[1])) {
          /* if a classification hypothesis can be formed on velocity */
          nostop = FALSE;                        /* set stopping condition */
          indic = YES;                           /* set formation indicator */
        }
        q = q->vnext;                            /* advance the pointer */
      }
      if (indic == YES)                          /* if a hypoth can be formed */
        csclhyp (p, r, 1);                       /* create hyp and set params */
    }
    p = p->fll;                                /* advance the pointer */
  }
  return;                                      /* return to point of call */
}

/* This routine is a production rule level 4 to level 5 */
1415r2 () {
  struct evnt *p;
  struct velist *q;
  struct araynod *r;
```

```
  int indic, nostop;
  p = evhdr->fll;                              /* point to 1st event node */
  while (p != evtl) {                          /* while not thru list, do */
    if (p->updtim == systime) {                /* if a current form or update */
      nostop = TRUE;                             /* set no stop indicator */
      indic = NO;                                /* set class hyp form marker */
      r = ((((p->lowr)->lfoll)->vslink)->lnk)->parent; /* get array ptr */
      if ((p->velmin < r->vmin[2]) && (p->velmax > r->vmax[2])) {
        /* if a classification hypothesis can be formed outright */
        nostop = FALSE;                          /* set stopping condition */
        indic = YES;                             /* set formation indicator */
      }
      else                                       /* otherwise */
        q = (p->vhdr)->vnext;                    /* prepare list search */
      while ((nostop) && (q != p->vsent)) {  /* while not stop, do */
        if ((q->vel >= r->vmin[2]) && (q->vel <= r->vmax[2])) {
          /* if a classification hypothesis can be formed on velocity */
          nostop = FALSE;                        /* set stopping condition */
          indic = YES;                           /* set formation indicator */
        }
        q = q->vnext;                            /* advance the pointer */
      }
      if (indic == YES)                          /* if a hypoth can be formed */
        csclhyp (p, r, 2);                       /* create hyp and set params */
    }
    p = p->fll;                                /* advance the pointer */
  }
  return;                                      /* return to point of call */
}

/* This routine is a production rule level 4 to level 5 */
1415r3 () {
  struct evnt *p;
  struct velist *q;
  struct araynod *r;
  int indic, nostop;
```

```
  p = evhdr->fll;                              /* point to 1st event node */
  while (p != evtl) {                          /* while not thru list, do */
    if (p->updtim == systime) {                /* if a current form or update */
      nostop = TRUE;                             /* set no stop indicator */
      indic = NO;                                /* set class hyp form marker */
      r = ((((p->lowr)->lfoll)->vslink)->lnk)->parent; /* get array ptr */
      if ((p->velmin < r->vmin[3]) && (p->velmax > r->vmax[3])) {
        /* if a classification hypothesis can be formed outright */
        nostop = FALSE;                          /* set stopping condition */
        indic = YES;                             /* set formation indicator */
      }
      else                                       /* otherwise */
        q = (p->vhdr)->vnext;                    /* prepare list search */
      while ((nostop) && (q != p->vsent)) {    /* while not stop, do */
        if ((q->vel >= r->vmin[3]) && (q->vel <= r->vmax[3])) {
          /* if a classification hypothesis can be formed on velocity */
          nostop = FALSE;                        /* set stopping condition */
          indic = YES;                           /* set formation indicator */
        }
        q = q->vnext;                            /* advance the pointer */
      }
      if (indic == YES)                          /* if a hypoth can be formed */
        csclhyp (p, r, 3);                       /* create hyp and set params */
    }
    p = p->fll;                                /* advance the pointer */
  }
  return;                                      /* return to point of call */
}

/* This routine is a production rule level 4 to level 5 */
1415r4 () {
  struct evnt *p;
  struct velist *q;
  struct araynod *r;
  int indic, nostop;
  p = evhdr->fll;                              /* point to 1st event node */
```

```
  while (p != evtl) {                          /* while not thru list, do */
    if (p->updtim == systime) {                /* if a current form or update */
      nostop = TRUE;                             /* set no stop indicator */
      indic = NO;                                /* set class hyp form marker */
      r = ((((p->lowr)->lfoll)->vslink)->lnk)->parent; /* get array ptr */
      if ((p->velmin < r->vmin[4]) && (p->velmax > r->vmax[4])) {
        /* if a classification hypothesis can be formed outright */
        nostop = FALSE;                          /* set stopping condition */
        indic = YES;                             /* set formation indicator */
      }
      else                                       /* otherwise */
        q = (p->vhdr)->vnext;                    /* prepare list search */
      while ((nostop) && (q != p->vsent)) {  /* while not stop, do */
        if ((q->vel >= r->vmin[4]) && (q->vel <= r->vmax[4])) {
          /* if a classification hypothesis can be formed on velocity */
          nostop = FALSE;                        /* set stopping condition */
          indic = YES;                           /* set formation indicator */
        }
        q = q->vnext;                            /* advance the pointer */
      }
      if (indic == YES)                          /* if a hypoth can be formed */
        csclhyp (p, r, 4);                       /* create hyp and set params */
    }
    p = p->fll;                                /* advance the pointer */
  }
  return;                                      /* return to point of call */
}

/* This routine is a production rule level 4 to level 5 */
1415r5 () {
  struct evnt *p;
  struct velist *q;
  struct araynod *r;
  int indic, nostop;
  p = evhdr->fll;                              /* point to 1st event node */
  while (p != evtl) {                          /* while not thru list, do */
```

```
    if (p->updtim == systime) {                 /* if a current form or update */
      nostop = TRUE;                              /* set no stop indicator */
      indic = NO;                                 /* set class hyp form marker */
      r = ((((p->lowr)->lfoll)->vslink)->lnk)->parent; /* get array ptr */
      q = (p->vhdr)->vnext;                       /* prepare list search */
      while ((nostop) && (q != p->vsent)) {       /* while not stop, do */
        if (q->vel >= r->vmin[5]) {               /* if a hyp can be formed */
          nostop = FALSE;                         /* set stopping condition */
          indic = YES;                            /* set formation indicator */
        }
        q = q->vnext;                             /* advance the pointer */
      }
      if (indic == YES)                           /* if a hypoth can be formed */
        csclhyp (p, r, 5);                        /* create hyp and set params */
    }
    p = p->fll;                                 /* advance the pointer */
  }
  return;                                       /* return to point of call */
}

/* This routine is a production rule level 5 to level 5 */
l515r0 () {
  struct classh *p, *temp;
  struct eventl *q;
  cltl->cupd = systime;                         /* set abs stopping condition */
  p = clhdr->hnext;                             /* point to 1st cl hyp node */
  while (p->cupd != systime) {                  /* while not thru list, do */
    q = (p->clower)->elnext;                    /* point to 1st list node */
    while ((q != p->chtl) && (q->elink != NIL))  /* while no stop, do */
      q = q->elnext;                            /* advance pointer */
    if (q != p->chtl) {                         /* if removal indicated */
      temp = p;                                 /* set temporary pointer */
      p = p->hprev;                             /* reset working pointer */
      remchyp (temp);                           /* remove the node */
    }
    p = p->hnext;                               /* advance the pointer */
```

```
    }
    return;                                    /* return to point of call */
}
```

Example Sensor Readout

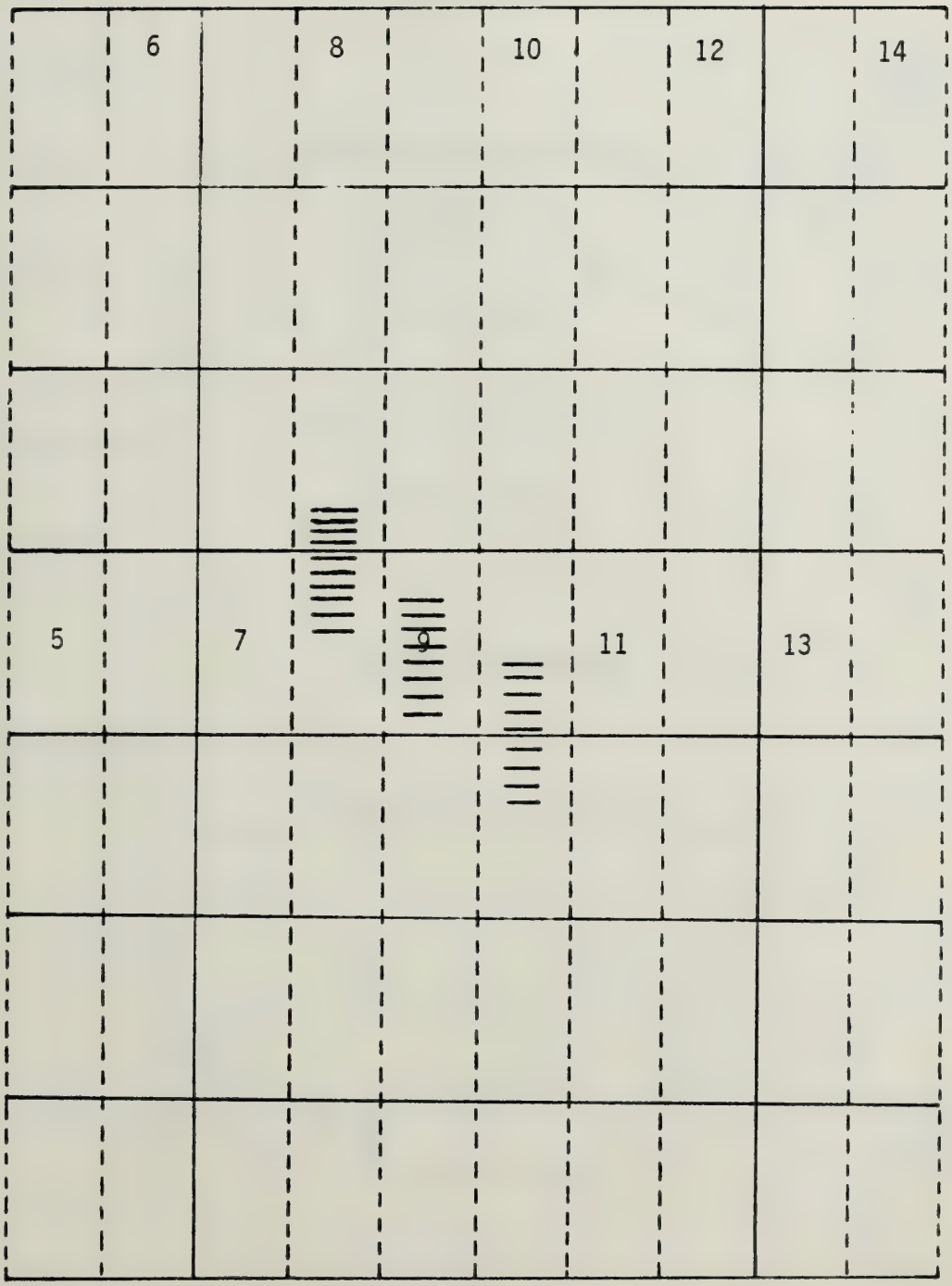
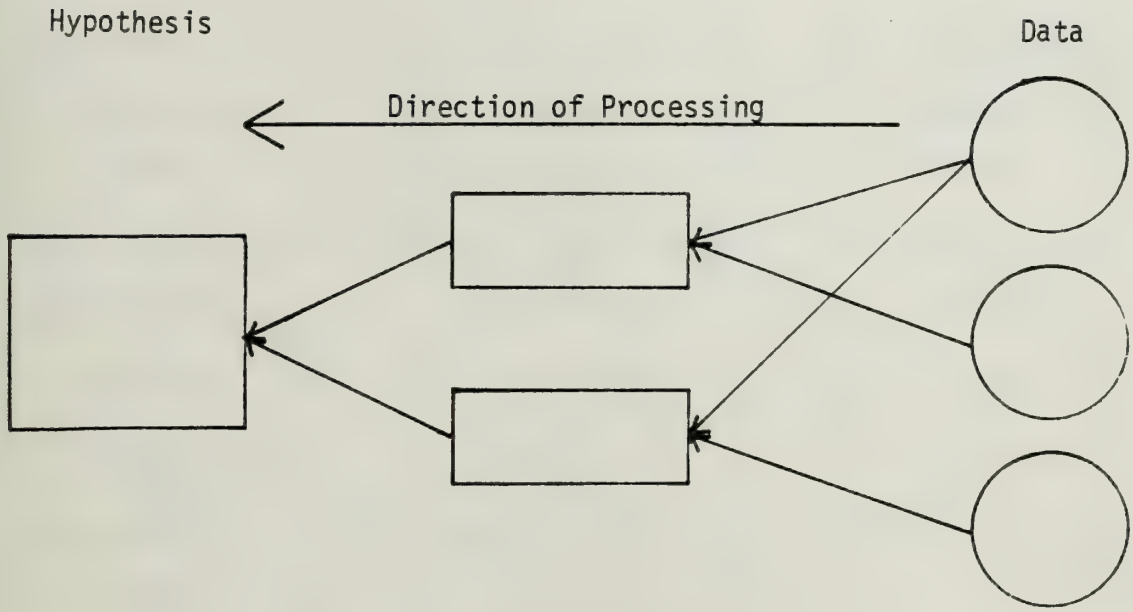


Figure 1

Processing Methods

Forward Chaining



Backward Chaining

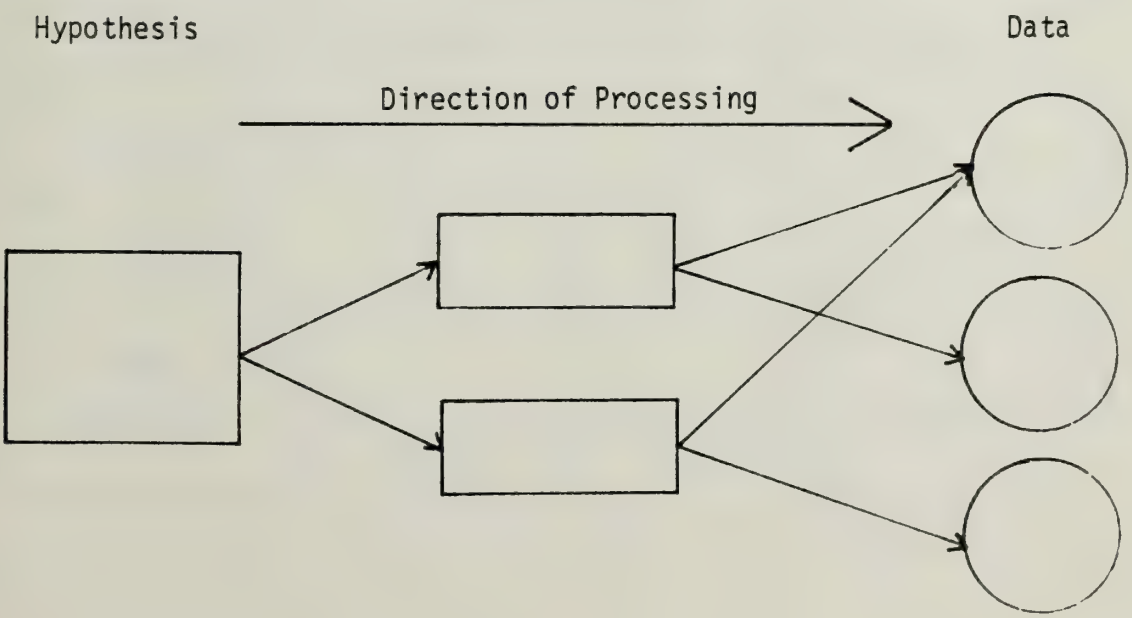


Figure 2

Overall System Design

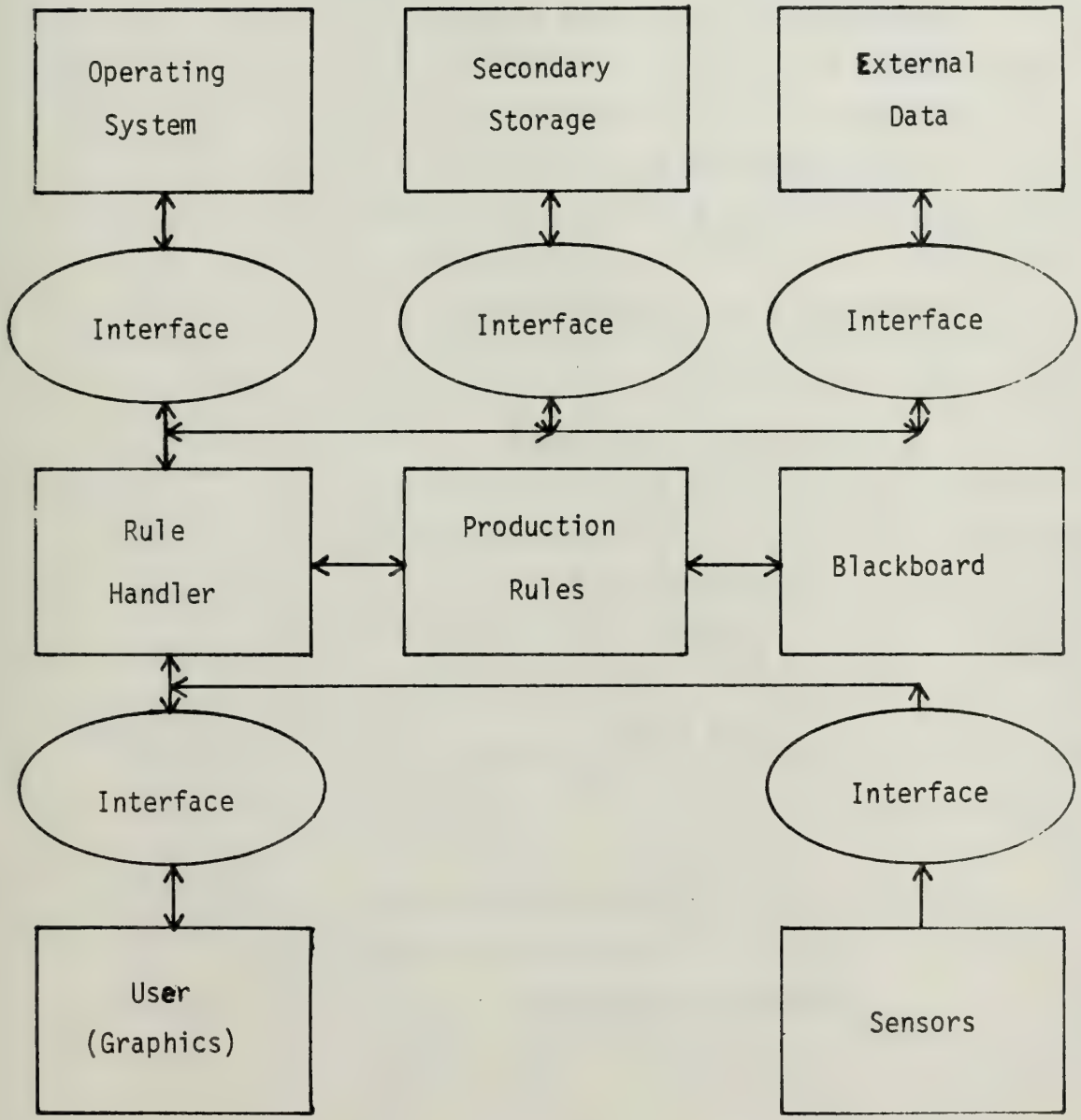


Figure 3

Production System Levels

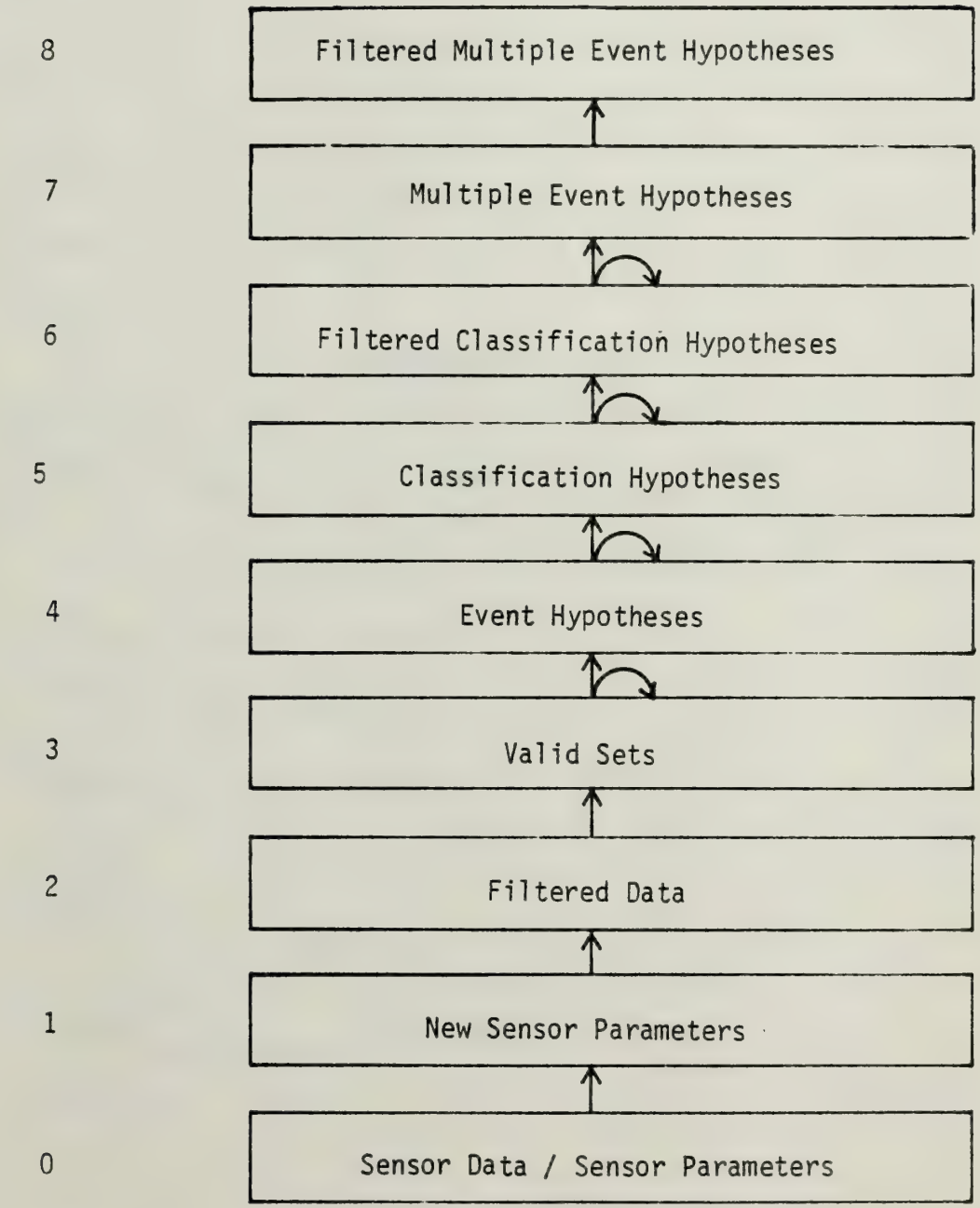


Figure 4

System Configuration Before Activation

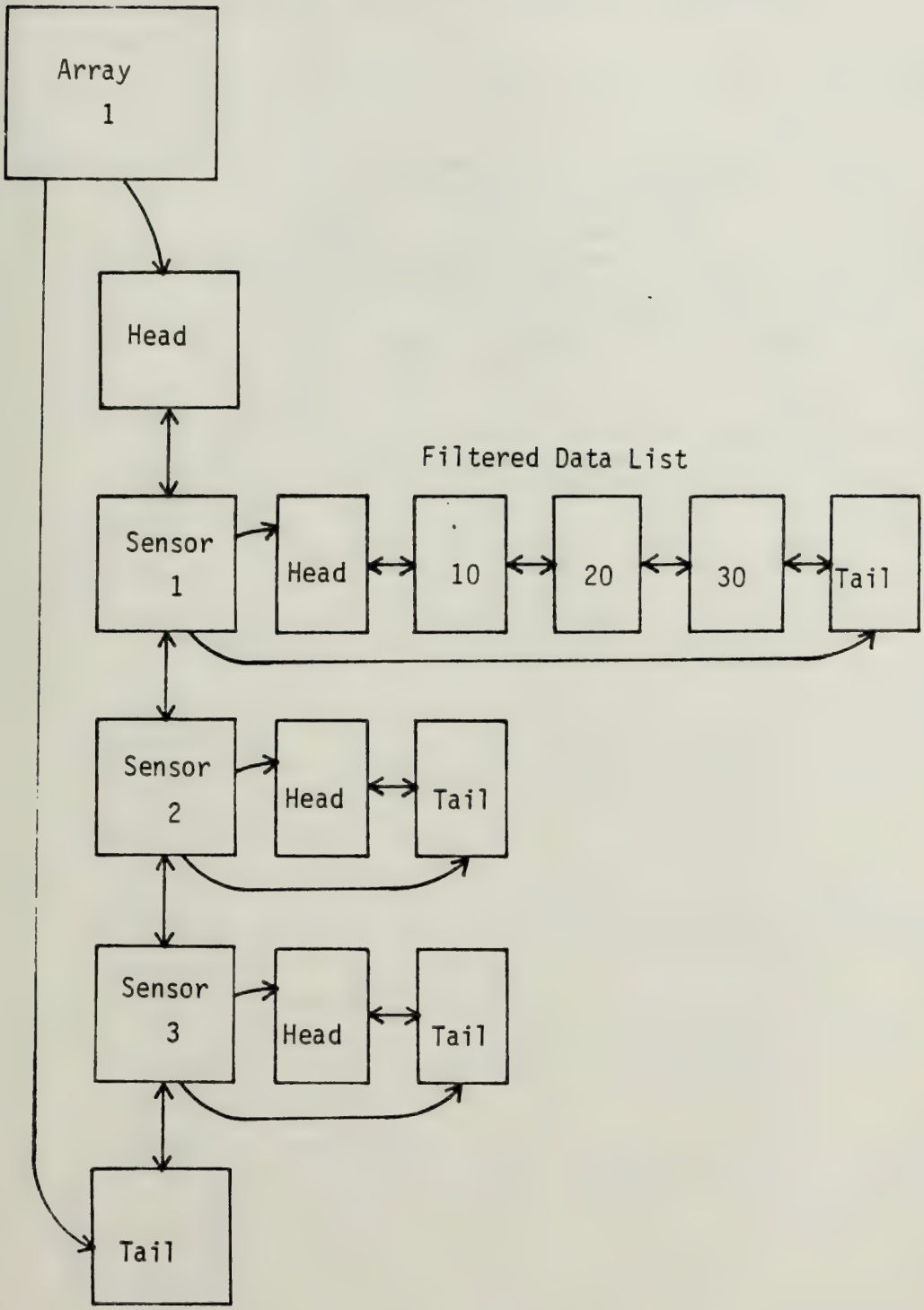


Figure 5

System Configuration After Activation

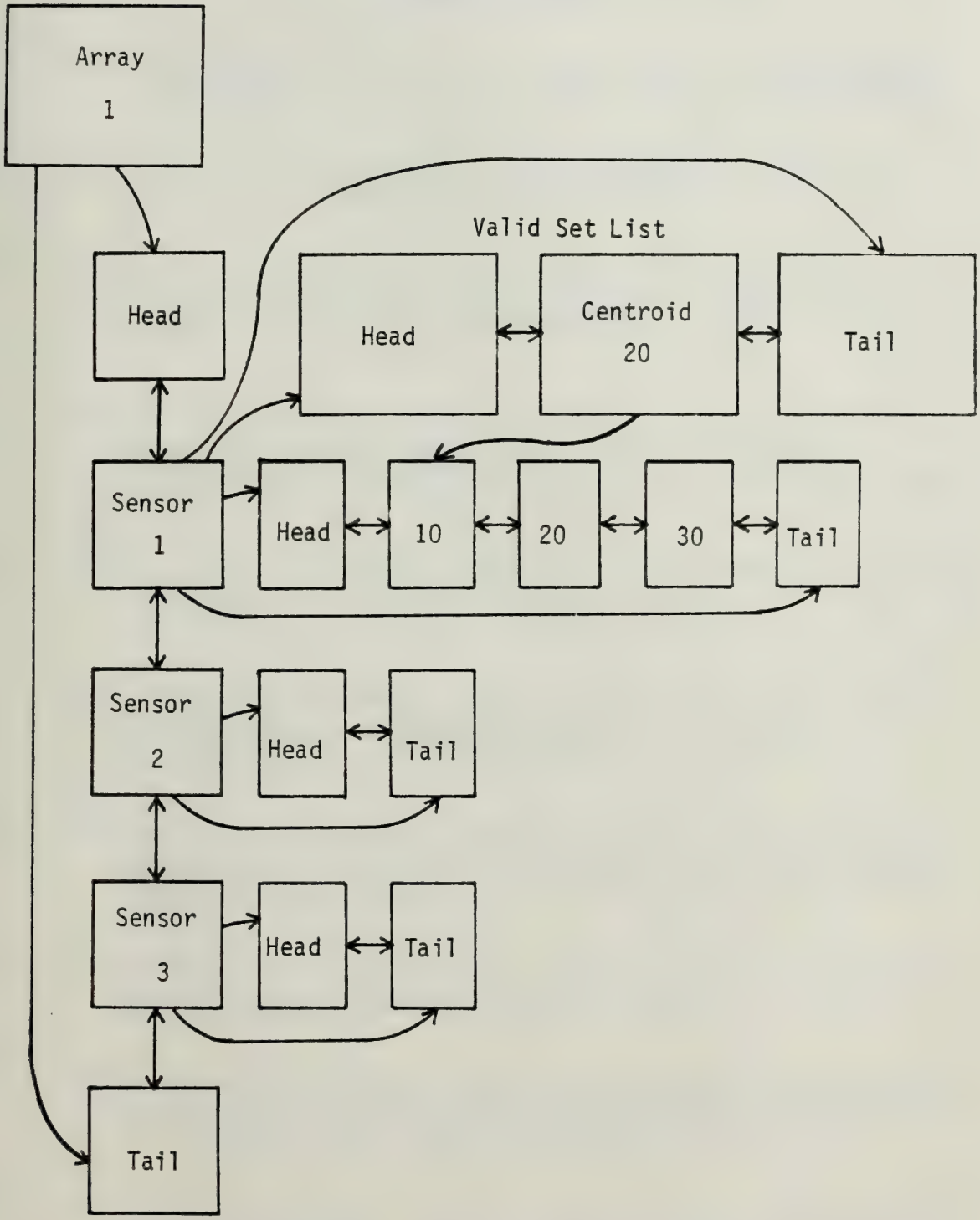


Figure 6

BIBLIOGRAPHY

1. United States Marine Corps, Fleet Marine Force Manual (FMFM) 2-1, p. 119-144.

2. Winston, P.H., Artificial Intelligence, p. 143-155, Addison-Wesley, 1979.

3. Davis, R. and King, J., "An Overview of Production Systems," Machine Intelligence, [v. 8], p. 300-331, 1976.

4. Marine Corps Development and Education Command DB 5-71, Tactical Surveillance Center/Tactical Sensor Surveillance (TSC/TSS) Concept, p. 1-15, July 1971.

5. Marine Corps Development and Education Command DB 4-71, Concept For the Employment of Sensors, p. 11-16, July 1971.

6. U.S. Army Intelligence Center and School SupR 17012-M, Interpretation and Analysis Text For Remote Sensors (RemS), October 1976.

7. U.S. Army Intelligence Center and School SupR 17011-M, Operational and Tactical Concepts For Employment of Remote Sensors (RemS), October 1976.

8. U.S. Army Intelligence Center and School SupR 17001-M, Remote Sensor Devices, October 1976.

9. Sandia Laboratory Report 0423, Human Factors Evaluation of the Terminal Message Processor (TMP), by M.N. Cravens, p. 4-20, July 1972.

10. U.S. Army Electronics Command Report 4403, An Automatic Target Tracking and Classification Algorithm (ATTAC) For Unattended Ground Sensors, by A.J. Farnochi, J. Karakowski, and E.D. Ulrich, p. 1-28, April 1976.

11. Lesser, V.R. and Erman, L.D., "A Retrospective View of the HEARSAY-II Architecture," Proceedings of the 5th International Joint Conference On Artificial Intelligence, p. 790-800, 1977.

12. Sandia Laboratory Report 0889, Criteria For Processing Activation Sequences From Seismic Sensors, by M.N. Cravens, p. 5-26, January 1972.

13. Nii, H.P. and Feigenbaum, E.A., "Rule-Based Understanding of Signals," Pattern Directed Inference Systems, p. 483-501, 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52Bz Computer Science Department Naval Postgraduate School Monterey, California 93940	2
4.	Prof. Harold Titus, Code 62Ts Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	2
5.	Prof. Douglas Smith, Code 52Sc Computer Science Department Naval Postgraduate School Monterey, California 93940	2
6.	Capt. Dennis M. Jackson, USMC 6233 27th Avenue North St. Petersburg, Florida 33710	2

Thesis
J213
c.1
189465
Jackson
Production rule system as an approach to interpretation of ground sensor information.
17 NOV 83
16 OCT 86
5 JUL 88
33348
32749

Thesis
J213
c.1
189465
Jackson
Production rule system as an approach to interpretation of ground sensor information.

thesJ213
Production rule systems as an approach t
3 2768 002 11013 2
DUDLEY KNOX LIBRARY